Le Programmeur

ActionScript™ 3.0 pour les jeux

Gary Rosenzweig

Couvre Flash® CS3

PEARSON

LE PROGRAMMEUR

ActionScript 3.0 pour les jeux

Gary Rosenzweig

CampusPress a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, CampusPress n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

CampusPress ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par CampusPress

47 bis, rue des Vinaigriers

75010 PARIS

Tél. : 01 72 74 90 00

Réalisation PAO : Léa B.

ISBN : 978-2-7440-4013-9
Copyright© 2009 Pearson Education France
Tous droits réservés

Titre original : *ActionScript 3.0 Game Programming University*

Traduit de l'américain par Patrick Fabre

ISBN original : 978-0-7897-3702-1

Copyright © 2008 by Que Publishing

All rights Reserved.

www.quepublishing.com

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Table des matières

Introduction 1

Flash et le développement de jeux 1
À qui ce livre est-il destiné ? 2
Que vous faut-il pour utiliser ce livre ? . . . 3
Utiliser les jeux d'exemple dans vos propres projets 4
Ce que vous trouverez dans ce livre. 4
Le site Web FlashGameU.com 5

Chapitre 1. Utiliser Flash et ActionScript 3.0 7

Qu'est-ce qu'ActionScript 3.0 ? 8
Créer un programme ActionScript simple . . . 9
Travailler avec Flash CS3 17
Écrire et modifier du code ActionScript. . . 21
Stratégies de programmation des jeux ActionScript 24
Notions élémentaires du langage ActionScript. 28
Test et débogage 32
Publier votre jeu 36
Chek-list de la programmation de jeux ActionScript 40

Chapitre 2. Composants de jeu ActionScript 45

Créer des objets visuels 46
Accepter les entrées de l'utilisateur 61
Créer une animation 66
Programmer l'interaction avec l'utilisateur . . 72
Accéder à des données externes 77
Éléments de jeu divers 80

Chapitre 3. Structure de jeu élémentaire : le Memory 91

Placer les éléments interactifs 93
Jeu 104
Encapsuler le jeu 112
Ajouter un score et un chronomètre 116
Ajouter des effets 123
Modifier le jeu 131

Chapitre 4. Jeux cérébraux : Simon et déduction 133

Tableaux et objets de données 134
Jeu de Simon 138
Jeu de déduction. 151

Chapitre 5. Animation de jeu : jeux de tir et de rebond 169

Animation de jeu 170
Air Raid 176
Casse-brique 195

Chapitre 6. Puzzles d'images et puzzles coulissants 211

Manipuler des images bitmap 212
Jeu de puzzle coulissant. 217
Puzzle classique 231
Chargement et découpage de l'image 234

Chapitre 7. Direction et mouvement : astéroïdes 245

Utiliser *Math* pour faire pivoter et déplacer des objets 246

Utiliser *Math* pour faire pivoter et déplacer des objets 249

Air Raid II 255

Space Rocks 262

Astéroïdes 279

Chapitre 8. Les "casual games" : Match Three 291

Classe réutilisable : Point Bursts 292

Match Three 301

Chapitre 9. Jeux de mots : pendu et mots mêlés 323

Chaînes et champs texte 324

Pendu 336

Mots mêlés 340

Chapitre 10. Questions et réponses : quiz et jeux de culture générale 357

Stocker et retrouver des données de jeu . . . 358

Quiz de culture générale 363

Quiz version Deluxe 375

Quiz d'images 385

Chapitre 11. Jeux d'action : jeux de plate-forme 393

Conception du jeu 394

Créer la classe 403

Modifier le jeu 425

Chapitre 12. Jeux de mondes : jeux de conduite et d'exploration 427

Créer un jeu de conduite en vue aérienne 428

Créer un jeu de course 451

Index 465

L'auteur

Enfant, Gary Rosenzweig est autorisé à jouer aux jeux vidéo autant qu'il le souhaite, pourvu qu'il ait d'abord terminé ses devoirs. Il adore jouer à Adventure, Asteroids, Pitfall, Raiders of the Lost Ark et même à l'épouvantable jeu d'E.T.

À 13 ans, en 1983, il reçoit de sa grand-mère un nouveau TRS-80 Model III. Il apprend immédiatement à programmer. Puis crée des jeux. Il conçoit des jeux d'aventure et de rôle avec du texte, puis des jeux d'arcade. On lui permet de passer ses nuits à créer des jeux, pourvu qu'il ait d'abord fait ses devoirs.

Au lycée, Gary s'amuse presque quand il le veut avec des ordinateurs Apple II, mais il commence toujours par faire ses devoirs. Il crée des simulateurs spatiaux, des programmes de tableur et des jeux.

Gary part étudier l'informatique à l'université de Drexel. On lui fait savoir qu'avec ce diplôme il peut être embauché comme programmeur dans n'importe quelle entreprise de logiciels pour entreprises. Mais lui veut créer des jeux, en guise de passe-temps, une fois son travail fait.

Après un détour qui lui vaut un diplôme de Master en journalisme et en communication de l'université de Caroline du Nord, à Chapel Hill, Gary prend un travail qui lui permet de créer des jeux pour les enfants avec le logiciel Director, de Macromedia.

Puis survient Internet. Rapidement après apparaît Shockwave, qui permet de lire du contenu Director dans des pages Web. Gary commence à créer ses propres jeux pour son site Web le soir, une fois son travail fait.

En 1996, Gary lance sa propre société, CleverMedia, afin de produire des jeux pour le Web. Il crée rapidement des jeux Shockwave et Flash avec certaines des personnes les plus créatives qu'il ait jamais rencontrées. CleverMedia et ses sites prennent de l'ampleur au fil des ans, jusqu'à devenir la plus grande collection de jeux Web d'une même entreprise. Gary crée plus de trois cents jeux au cours des douze dernières années, dont la plupart se trouvent sur le site principal de CleverMedia, **www.GameScene.com**.

Gary aime aussi partager ce qu'il sait. Ses sites **http://FlashGameU.com**, **www.Director-Online.com** et **www.DeveloperDispatch.com** contiennent des informations pour les autres développeurs. Il écrit de nombreux livres, dont *Macromedia Flash MX ActionScript for Fun & Games*, *Special Edition Using Director MX* et *Advanced Lingo for Games*. Gary a écrit ce livre en grande partie pendant ses soirées et ses week-ends, une fois son travail fait.

Gary vit à Denver, dans le Colorado, avec sa femme, Debby, et sa fille, Luna. Debby et Gary possèdent également la librairie Attic Bookstore, une échoppe incongrue de livres d'occasion à Englewood, dans le Colorado. Luna n'a que 5 ans mais joue déjà à des jeux sur son ordinateur Macintosh. On l'y autorise mais, bien sûr, une fois ses devoirs faits.

Introduction

Voici un moment bien choisi pour devenir un développeur de jeux Flash. Aujourd'hui, il n'existe pas de meilleur outil de développement pour les jeux de petite et de moyenne taille.

Flash CS3 Professional (aussi appelé Flash 9) est rapide, puissant et très simple d'usage pour les projets de développement. La clé de ce succès tient à ActionScript 3.0, l'excellent nouveau langage de programmation livré dans la dernière version de Flash.

ActionScript 1.0 et 2.0 se sont souvent révélés frustrants pour les développeurs de jeux. Ils n'étaient pas assez rapides pour la réalisation des tâches essentielles, et d'étranges bogues ou des comportements inattendus venaient souvent compliquer la production.

ActionScript 3.0 est d'un tout autre acabit. Il permet de développer des programmes rapidement et sans effort. Tout fonctionne et fonctionne bien. La vitesse d'ActionScript 3.0 permet en outre d'implémenter des solutions exactement comme on les a imaginées.

Ce livre doit devenir votre guide pour le développement des jeux Flash. J'espère que vous apprécierez autant d'apprendre en lisant ces pages que j'ai pris plaisir à les écrire.

Flash et le développement de jeux

En octobre 1995, j'étais très excité par mon avenir en tant que développeur de jeux. Macromedia venait de lancer Shockwave et je découvrais qu'il s'agissait d'un moyen de développer des jeux que je pourrais distribuer moi-même sur le Web.

Je n'ai retrouvé un tel sentiment d'excitation concernant le développement des jeux que deux fois depuis la sortie du premier Shockwave. La première fut à la sortie de Shockwave 3D. La seconde, lors de la sortie d'ActionScript 3.0.

Cela fait un certain temps que les jeux Flash ont fait leur entrée sur le devant de la scène, mais ils n'y ont jamais figuré que comme une sorte de succédané des jeux Shockwave. Shockwave était plus rapide, plus puissant et proposait même des fonctionnalités 3D.

Avec ActionScript 3.0, Flash atteint la puissance de Shockwave et la dépasse même sur certains points. Par exemple, le lecteur Flash 9 est déjà disponible sur 80 % des ordinateurs qui surfent sur le Web. D'ici à ce que vous lisiez ce livre, la plupart des lecteurs Flash 8 auront été mis à jour avec la version 9 et nous nous rapprocherons d'un taux effectif de 100 %. Le fait de savoir que Flash 9 est presque aussi répandu que les navigateurs Web eux-mêmes ouvre une tout autre dimension pour les développeurs de jeux Flash.

Flash 9 s'exécute même sur les ordinateurs Linux. Les anciennes versions de Flash s'exécutent sur les consoles de TV Web, les consoles de jeu comme la Wii et même les périphériques portables tels que les smartphones ou la PlayStation Portable. Le lecteur Flash 9 et ActionScript 3.0 finiront d'ailleurs par être disponibles sur ces types de périphériques également.

Vous pouvez développer des versions autonomes ou des versions Web de vos jeux avec Flash. Des logiciels tiers vous permettent d'étendre les jeux autonomes afin de les transformer en de véritables applications.

Flash constitue avec ActionScript 3.0 un excellent moyen de créer des jeux de petite et de moyenne taille.

À qui ce livre est-il destiné ?

Ce livre est destiné à tous ceux qui utilisent Flash pour développer des jeux. Tous les développeurs n'utiliseront cependant pas ce livre de la même manière.

Ceux qui débutent avec Flash et la programmation pourront en faire leur seconde étape après une première familiarisation avec les notions de base de la programmation. Un débutant motivé peut utiliser ce livre pour apprendre le langage ActionScript 3.0 de toutes pièces.

Si vous possédez déjà une expérience en programmation avec ActionScript 1.0 ou 2.0, vous pourrez utiliser ce livre pour vous mettre à la page avec ActionScript 3.0.

Il est néanmoins préférable d'oublier l'essentiel de ce que vous avez appris avec les précédentes versions du langage. ActionScript 3.0 est considérablement différent. Je considère même pour ma part qu'il s'agit d'un langage de programmation entièrement nouveau.

Bon nombre d'utilisateurs Flash connaissent déjà les rudiments de l'animation et de la programmation mais souhaitent maintenant développer des jeux. Voici le premier public ciblé par ce livre.

Si vous êtes non pas programmeur mais concepteur, illustrateur ou animateur, vous pourrez utiliser les exemples de ce livre comme structure pour vos propres jeux. Autrement dit, vous pourrez vous contenter de remplacer les graphismes dans les fichiers source d'exemple.

De la même manière, si vous êtes déjà un programmeur ActionScript 3.0 averti, vous découvrirez dans ce livre une bibliothèque de code dans laquelle vous pourrez puiser pour vos propres jeux. Inutile de réinventer la roue.

Que vous faut-il pour utiliser ce livre ?

La plupart des lecteurs devront avoir une expérience préalable dans l'utilisation de Flash et en programmation pour tirer le meilleur parti de ce livre. Vous devrez aussi posséder les bons outils.

Connaissances requises

Les lecteurs doivent être familiarisés avec l'environnement de Flash CS3. Si vous découvrez Flash, commencez par parcourir le Guide de l'utilisateur Flash livré avec le logiciel. Dans Flash, choisissez Aide > Aide Flash ou appuyez sur F1. Il peut aussi être utile de se procurer un livre pour les débutants ou de consulter des didacticiels en ligne.

Ce livre n'est pas destiné aux programmeurs débutants, à moins que vous ne cherchiez à utiliser les exemples en y insérant vos propres graphismes. Vous devez donc déjà avoir une certaine expérience en programmation, que ce soit avec ActionScript 1.0, 2.0 ou 3.0, JavaScript, Java, Lingo, Perl, PHP, C++ ou n'importe quel autre langage de programmation structuré. ActionScript 3.0 n'est pas difficile à comprendre si vous avez l'habitude des boucles, des instructions conditionnelles et des fonctions. Le Chapitre 1 propose même une introduction à la syntaxe de base d'ActionScript 3.0.

Si vous êtes programmeur mais que vous n'ayez jamais utilisé Flash auparavant, lisez les parties du Guide de l'utilisateur qui concernent l'interface Flash et les techniques de dessin et d'animation élémentaires.

Applications logicielles

Vous aurez évidemment besoin de Flash CS3 Professional ou d'une version ultérieure. Flash 8 Studio, la précédente version de Flash, n'utilise pas ActionScript 3.0 et ne peut donc être utilisée avec ce livre.

Flash CS3 est virtuellement identique sur Mac et sous Windows. Les captures d'écran de ce livre, qu'elles proviennent de la version Mac ou Windows de Flash, devraient donc ressembler en tous points à ce que vous obtiendrez sur votre propre ordinateur.

Les futures versions de Flash continueront selon toute vraisemblance à utiliser ActionScript 3.0 comme langage de programmation principal. Certains des choix de menu et des raccourcis clavier pourraient changer, mais vous devriez malgré tout pouvoir utiliser ce livre. Il peut être utile dans ce cas de régler vos paramètres de publication en sélectionnant le lecteur Flash 9 et ActionScript 3.0 pour assurer une compatibilité maximale.

Fichiers source

Vous aurez également besoin des fichiers source de ce livre. Consultez la fin de l'introduction pour plus d'informations sur la manière de les acquérir.

Utiliser les jeux d'exemple dans vos propres projets

Ce livre inclut seize jeux complets, dont de petits bijoux comme Match Three, un jeu de plate-forme défilant et un jeu de mots mêlés. On me pose souvent la question : "Est-ce que je peux utiliser ces jeux dans mon projet ?"

La réponse est oui, pour autant que vous les modifiiez afin de vous les approprier, par exemple en adaptant les graphismes, en changeant les modalités de jeu ou en modifiant d'autres éléments du contenu. Ne postez pas les jeux en l'état sur votre site Web. Il n'est pas non plus toléré que vous publiiez le code source ou les listings du code de ce livre afin de les mettre à la disposition du public.

Lorsque vous utiliserez ces jeux dans vos projets, ne faites pas comme s'il s'agissait entièrement de votre propre travail. Ce ne serait pas professionnel. Merci de mentionner ce livre en proposant un lien vers **http://flashgameu.com**.

En revanche, si vous n'utilisez qu'une petite portion de code ou si vous utilisez un jeu comme structure de base pour l'élaboration d'un jeu complètement différent, il n'est pas nécessaire d'ajouter de mention particulière. Tout n'est finalement qu'affaire de bon sens et de courtoisie. Je vous en remercie d'avance.

Ce que vous trouverez dans ce livre

Le Chapitre 1, "Utiliser Flash et ActionScript 3.0", introduit ActionScript 3.0 et certains concepts élémentaires comme les stratégies de programmation de jeux et une check-list pour vous aider à développer vos jeux avec Flash CS3.

Le Chapitre 2, "Éléments de jeu ActionScript", présente une série de courts fragments de code et de fonctions permettant par exemple de créer des champs texte, de dessiner des formes et de lire des sons. Il s'agit d'une bibliothèque de code utile et pratique que nous utiliserons dans le reste du livre et que vous pourrez utiliser dans vos propres projets.

Les Chapitres 3 à 12 contiennent chacun un ou plusieurs jeux complets. Le texte des chapitres vous guide lors du développement du code du jeu, en vous permettant de le reproduire par vous-même si vous le souhaitez. Vous pouvez aussi utiliser les fichiers source et en consulter directement le code final.

Le Chapitre 3, "Structure de jeu élémentaire : un jeu de Memory", est un peu différent du reste du livre. Au lieu d'examiner le code d'un jeu fini, il construit le jeu en dix étapes, en produisant une animation Flash et un fichier de code source à chaque étape. Il s'agit d'un excellent moyen d'apprendre à construire des jeux Flash.

L'essentiel du reste des chapitres introduit un sujet spécial avant de commencer un nouveau jeu. Par exemple, le Chapitre 4 commence par une section "Tableaux et objets de données".

Le contenu de ce livre ne se limite pas aux pages que vous tenez entre vos mains. Il reste bien des choses à découvrir en ligne.

Le site Web *FlashGameU.com*

Le site **FlashGameU.com** est le site Web d'accompagnement de ce livre (en anglais). Vous pouvez vous y rendre pour trouver les fichiers source, les mises à jour, le nouveau contenu, un forum de développement de jeux Flash et, enfin, mon blog et mon podcast consacrés au développement de jeux avec Flash.

Les fichiers source du livre sont organisés par chapitre puis divisés en archives pour chaque jeu. Un lien est proposé pour télécharger les fichiers dans la page principale de **FlashGameU.com**.

Sur le site **FlashGameU.com**, vous trouverez également un blog (en anglais) sur lequel je poste le nouveau contenu et tente de répondre aux questions des lecteurs. Si vous avez une question concernant ce livre ou le développement des jeux Flash en général, posez-la sur le forum ou directement sur mon blog. À très bientôt !

1

Utiliser Flash et ActionScript 3.0

Au sommaire de ce chapitre

- Qu'est-ce qu'ActionScript 3.0 ?
- Créer un programme ActionScript simple
- Travailler avec Flash CS3
- Écrire et éditer du code ActionScript
- Stratégies de programmation de jeux ActionScript
- Concepts ActionScript élémentaires
- Test et débogage
- Publier votre jeu
- Check-list de programmation des jeux ActionScript

ActionScript est un langage de programmation parfaitement adapté à la création de jeux. Il est facile à apprendre, rapide à développer et très puissant.

Nous commencerons par examiner ActionScript 3.0 et l'environnement de création Flash CS3. Ensuite, nous construirons quelques programmes très simples afin de nous familiariser avec cette nouvelle version d'ActionScript.

Qu'est-ce qu'ActionScript 3.0 ?

ActionScript 3.0 a été introduit en 2006 à la sortie de Flex 2. Comme Flash, Flex permet aux développeurs de créer des applications qui utilisent le lecteur Flash Player. Flash propose cependant pour le développement des applications une interface plus visuelle qui convient particulièrement mieux au développement des jeux.

ActionScript a été introduit en 1996 à la sortie de Flash 4. Le langage ne s'appelait pas encore ActionScript et il n'était pas encore possible de taper du code. Il fallait choisir des instructions parmi une série de menus déroulants.

En 2000, Flash 5 a considérablement amélioré les choses grâce à l'introduction formelle d'ActionScript 1.0. Ce langage de script incluait tous les éléments dernier cri des autres langages de développement Web, comme le Lingo du logiciel Director de Macromedia et le Java de Sun. Il était cependant très limité en termes de vitesse et de puissance.

Flash MX 2004, aussi appelé Flash 7, introduisit ActionScript 2.0, une version bien plus puissante du langage qui facilitait la création de programmes orientés objet. Cette mouture s'apparentait plus à ECMA Script, un standard des langages de programmation développé par l'European Computer Manufacturers Association. JavaScript, le langage de programmation utilisé dans les navigateurs, est également élaboré à partir d'ECMA Script.

Le lecteur Flash Player 9 intègre deux interpréteurs de code séparés. Le premier est destiné au contenu plus ancien et interprète le code ActionScript 1.0/2.0. Le second est uninterpréteur plus rapide qui fonctionne avec ActionScript 3.0. Vous obtiendrez de meilleures performances dans vos jeux si vous n'utilisez que du code ActionScript 3.0.

ActionScript 3.0 est l'aboutissement de longues années de développement. À chaque nouvelle version du programme Flash, les développeurs lui ont fait atteindre de nouvelles limites. La dernière version à ce jour tient compte de l'usage que les développeurs font de Flash aujourd'hui et corrige les faiblesses de la précédente variante du langage.

Nous disposons maintenant d'un excellent environnement de développement pour la création des jeux en deux dimensions. Vous verrez que l'une de ses principales forces tient à sa capacité à élaborer des jeux parfaitement opérationnels avec très peu de code.

Flash CS3 Professional correspond en fait à Flash 9. Adobe a regroupé ensemble les versions de différents composants logiciels comme Flash, Photoshop, Illustrator et Dreamweaver dans la solution "CS3". Le numéro de version technique de Flash dans CS3 est Flash 9. On peut donc aussi bien faire référence à Flash 9 qu'à Flash CS3. Le moteur de lecture qui est également utilisé par Flex est quant à lui appelé Flash Player 9 uniquement.

Créer un programme ActionScript simple

Codes sources

http://flashgameu.com

A3GPU01_HelloWorld.zip

Lors de l'introduction d'un nouveau langage de programmation, la tradition veut que l'on commence par des programmes Hello World. L'idée consiste à écrire un programme simple qui se contente d'afficher les mots Hello World à l'écran.

Le programme Hello World remonte à 1974, lorsqu'il fut inclus dans un document didacticiel interne de Bell Labs. C'est le premier programme que j'ai appris à programmer en m'installant devant un terminal PDP-11 à l'école à la fin des années 1970. Presque tous les livres de programmation pour débutants commencent par un exemple Hello World.

Utilisation simple de *trace*

Une version simple du programme Hello World peut être créée en utilisant la fonction `trace` dans un script du scénario principal. `trace` se contente d'afficher du texte dans le panneau Sortie de Flash.

Pour créer une nouvelle animation Flash, choisissez Fichier > Nouveau dans la barre des menus. La fenêtre Nouveau document apparaît (voir Figure 1.1).

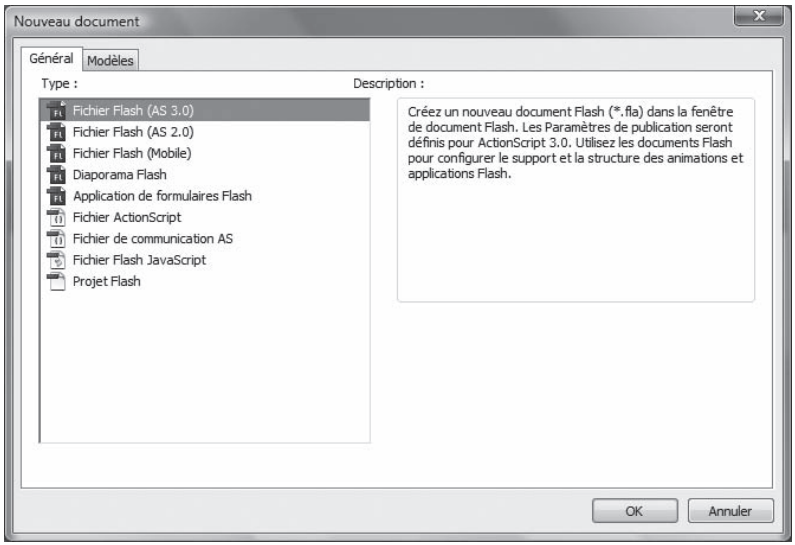


Figure 1.1
Choisissez Fichier Flash (AS 3.0) pour créer une nouvelle animation Flash.

Lorsque vous cliquez sur le bouton OK, vous accédez à une nouvelle animation Flash intitulée Sans nom-1. Cette animation s'affiche sous la forme d'une fenêtre Document Flash (voir Figure 1.2).

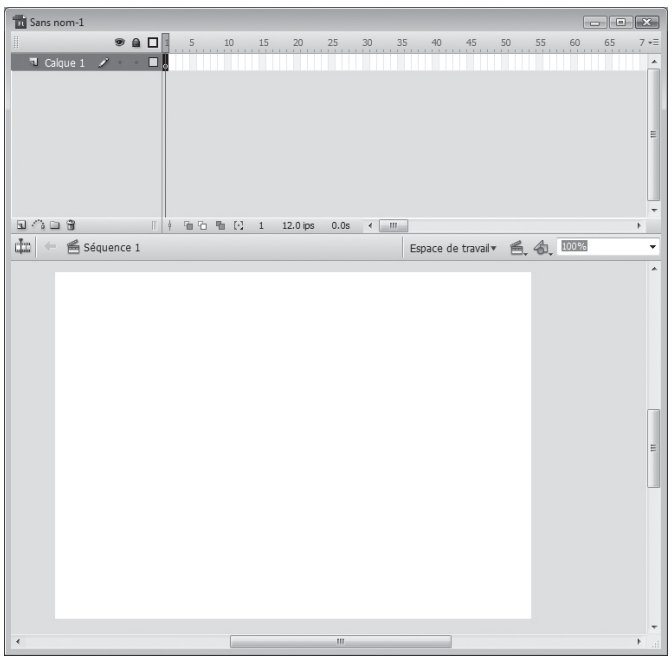


Figure 1.2
La fenêtre Document Flash inclut un scénario et une zone de travail de scène.

La partie supérieure de la fenêtre Document inclut un scénario, avec des images dont la première porte le numéro 1 et qui s'étendent vers la droite. La Figure 1.2 en révèle un peu plus de 65, mais ce nombre dépend de la taille de la fenêtre. Le nombre d'images peut être aussi important que le requiert l'animateur mais, pour la programmation de nos jeux, nous n'en utiliserons généralement que quelques-unes.

Le scénario peut contenir un ou plusieurs calques. Par défaut, il n'en existe qu'un, nommé Calque 1 dans la fenêtre Document.

Dans le Calque 1 apparaît une unique image-clé représentée par un rectangle marqué d'un petit cercle sous le numéro d'image 1.

Le terme image-clé est un terme d'animation. Si nous nous occupions d'animer des éléments dans Flash au lieu de les programmer, nous ne cesserions d'utiliser des images-clés. L'image-clé désigne un point du scénario où les positions d'un ou de plusieurs des éléments animés sont fixées de manière spécifique. Entre les images-clés, les éléments changent de position. Par exemple, si une image-clé à l'image 1 contient un élément à gauche de l'écran et une autre à l'image 9 contient le même élément à droite de l'écran, vous verrez l'élément au milieu de l'écran entre ces images-clés, à l'image 5.
Nous n'utiliserons pas d'image-clé pour créer des animations mais le ferons pour placer des éléments à l'écran dans différents modes : Intro, Jeu ou Fin de partie.

Vous pouvez placer un script dans n'importe quelle image-clé de n'importe quel calque du scénario. Pour cela, sélectionnez l'image-clé, choisissez le menu Fenêtre et sélectionnez Actions.

Le panneau Actions s'affiche (voir Figure 1.3). Il est possible qu'il possède une autre apparence sur votre écran, car il peut être personnalisé de nombreuses manières, notamment en affichant un ensemble complet de commandes et de fonctions ActionScript dans un menu à gauche.

Le panneau Actions est une simple fenêtre de saisie de texte. Il est cependant capable de bien des choses et peut par exemple vous aider à formater votre code. Nous n'utiliserons pas souvent ce panneau dans ce livre, car l'essentiel de notre code sera placé dans des classes externes.

Pour créer ce simple programme Hello World, tapez le texte suivant dans le panneau Actions :

```
trace("Hello World");
```

Voilà tout. Vous venez de créer votre premier programme ActionScript 3.0. Pour le tester, choisissez Contrôle > Tester l'animation ou tapez le raccourci Ctrl+Entrée. Si vous n'avez pas vous-même créé l'animation, ouvrez **HelloWorld1.fla** et utilisez ce fichier pour le test.

À présent, examinez le panneau Sortie. Celui-ci s'affiche même s'il était auparavant fermé. Comme il est souvent petit, il se peut toutefois qu'il apparaisse dans un coin de l'écran sans que vous ne le remarquiez. La Figure 1.4 montre à quoi il devrait ressembler à présent.

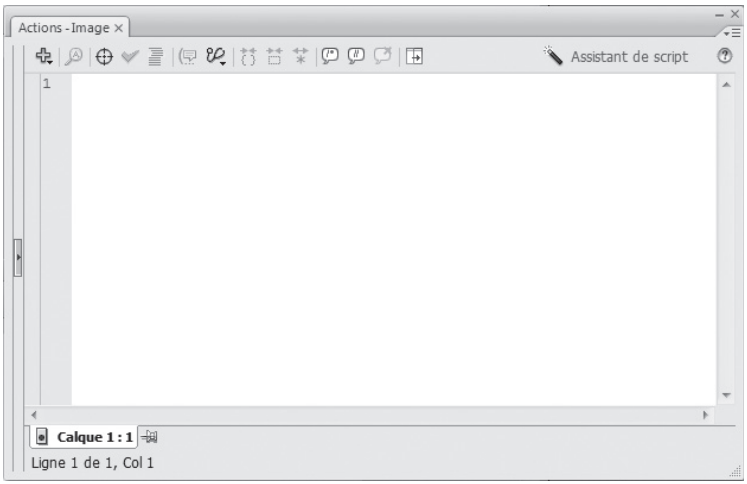


Figure 1.3
Le panneau Actions peut aussi être affiché à l'aide du raccourci clavier Alt + F9 *(Windows) ou* Alt + F9 *(Mac).*

Figure 1.4
Le panneau Sortie présente le résultat de l'appel de la fonction `trace`.

Si ce programme Hello World a techniquement pour effet d'afficher la chaîne "Hello World", il ne le fait que lorsque vous testez l'animation dans Flash 9. Si vous incorporiez cette animation dans un navigateur, vous ne verriez rien à l'écran. Quelques efforts supplémentaires sont ainsi nécessaires pour créer un véritable programme Hello World.

Créer une sortie écran

Pour que les mots Hello World s'affichent à l'écran, il nous faut plus d'une ligne de code. En réalité, il en faut trois.

La première créera une nouvelle zone de texte à afficher à l'écran que l'on appelle un champ texte. Il s'agit d'un conteneur pour du texte.

La deuxième ligne placera les mots Hello World dans ce champ texte.

La troisième ajoutera ce champ à la scène. La scène est la zone d'affichage des animations Flash. Vous pouvez disposer les éléments dans la scène pendant que vous créez une animation. À la lecture de l'animation, c'est la scène que voit l'utilisateur.

Avec ActionScript 3.0, le fait de créer des objets tels qu'un champ texte ne les ajoute pas à la scène. Cette opération doit être effectuée par vos soins. Ce principe de fonctionnement se révélera utile par la suite lorsque vous souhaiterez regrouper des objets sans les placer tous directement sur la scène.

Dans ActionScript 3.0, tous les éléments visuels correspondent à des types d'objets d'affichage. Il peut s'agir de champs texte, d'éléments graphiques, de boutons ou même de composants d'interface utilisateur (par exemple des menus déroulants). Les objets d'affichage peuvent également être des collections d'autres objets d'affichage. Par exemple, un objet d'affichage peut contenir toutes les pièces d'un jeu d'échecs et l'échiquier, figurer dans un autre objet d'affichage situé en dessous. La scène elle-même est un objet d'affichage ; il s'agit en fait d'un objet d'affichage appelé clip.

Voici les trois lignes de code de notre nouveau programme Hello World. Elles remplacent la ligne de code de l'image 1 du scénario de l'exemple précédent :

```
var myText:TextField = new TextField();
myText.text = "Hello World";
addChild(myText);
```

Le code crée une variable nommée `myText` de type `TextField`. Il attribue ensuite la valeur `"Hello World"` à la propriété `text` de ce champ texte avant de l'ajouter comme enfant à l'objet d'affichage qui correspond à la scène.

Le mot-clé `var` avant la première utilisation de la variable `myText` indique au compilateur que nous allons créer une variable nommée `myText`. Le signe deux points et le type `TextField` indiquent au compilateur le type de valeur que cette variable contiendra (dans le cas présent, une référence à un champ texte).

Ce programme produit un tout petit "Hello World" dans la police serif par défaut tout en haut à gauche de l'écran. Choisissez Contrôle > Tester l'animation pour le vérifier par vous-même. Le fichier source est **HelloWorld2.fla**. La Figure 1.5 présente le champ texte que nous venons de créer.

Le texte apparaît en haut à gauche et s'affiche avec cette police particulière parce que nous n'avons défini aucune autre propriété du champ texte. Lorsque nous en aurons appris un peu plus, nous pourrons choisir l'emplacement du texte, sa taille et sa police.

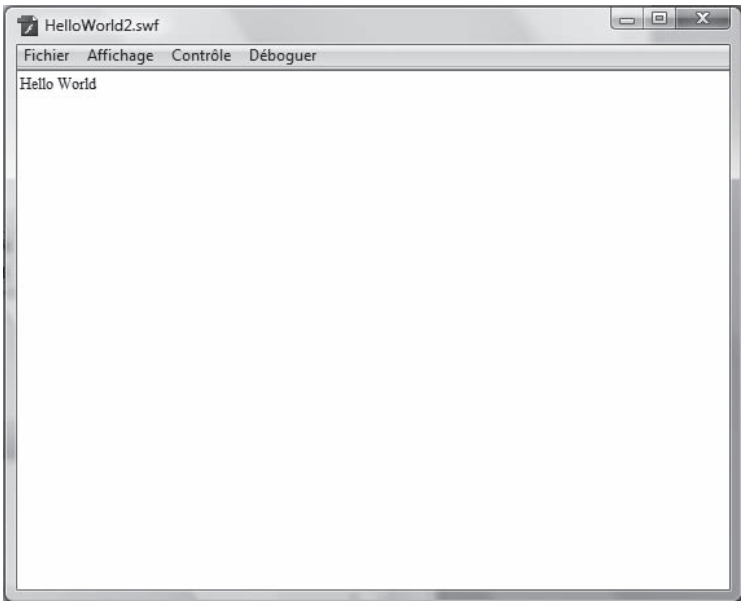


Figure 1.5
L'écran affiche un petit "Hello World" en haut à gauche.

Notre première classe ActionScript 3.0

Nous n'utiliserons pas de script dans le scénario à moins que nous n'ayons quelque chose de spécifique à réaliser dans une image donnée du scénario. Pour l'essentiel, notre code se trouvera placé dans des fichiers de classe ActionScript externe.

Recréons donc notre programme Hello World sous forme de classe externe.

Le terme classe est un autre moyen de faire référence à un objet Flash, que ce soit un élément graphique ou l'animation elle-même. Nous ferons aussi souvent référence à des classes pour désigner la portion de code d'un objet. Vous aurez ainsi une animation et la classe de cette animation. Cette dernière définira les données associées à l'animation et les fonctions qu'elle sera susceptible de réaliser.

Pour créer un fichier ActionScript externe, choisissez Fichier > Nouveau et sélectionnez Fichier ActionScript. Une nouvelle fenêtre Document ActionScript s'ouvre et vient occuper le même espace que la fenêtre Document de l'animation Flash. Au lieu d'un scénario et d'une scène, seule une grande zone d'édition de texte apparaît (voir Figure 1.6).

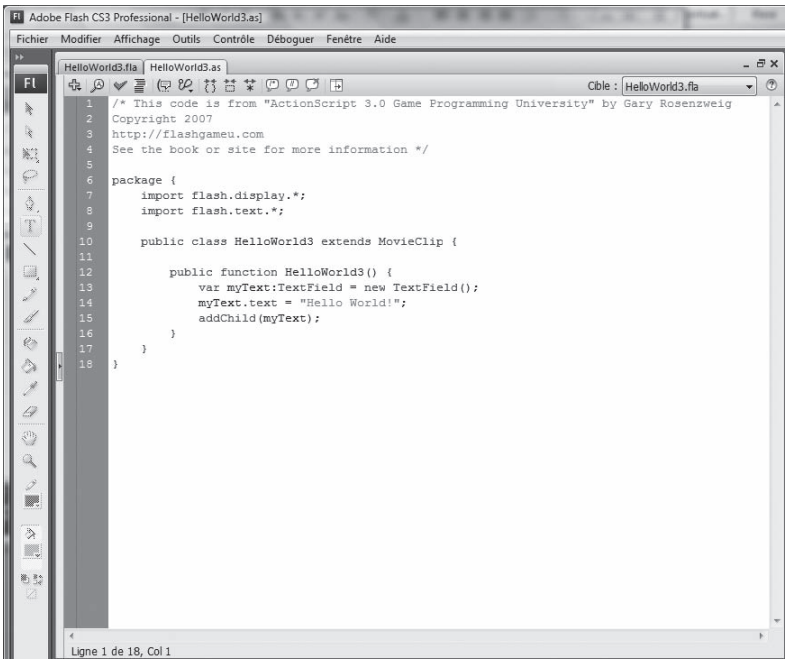


Figure 1.6
Le document ActionScript contient un programme Hello World très simple.

Comme vous pouvez le voir à la Figure 1.6, ce programme est bien plus long que le programme Hello World de trois lignes que nous avons créé précédemment. Voyons ce que réalise chacune des parties du code.

Le fichier de classe commence par déclarer qu'il s'agit d'un paquetage contenant une classe. Ensuite, il définit les parties d'ActionScript requises dans le programme. Dans le cas présent, nous devons afficher des objets sur la scène et créer un champ texte. Pour cela, nous devons utiliser les classes `flash.display` et `flash.text` :

```
package {
    import flash.display.*;
    import flash.text.*;
```

Vous apprendrez rapidement quelles classes de bibliothèque vous devez importer au début de vos programmes. Il ne s'agit là que de deux des bibliothèques parmi la multitude que nous utiliserons dans ce livre. Pour les fonctions ActionScript plus inhabituelles, vous pourrez toujours examiner l'entrée de la fonction correspondante dans l'Aide Flash 9 afin de voir quelle bibliothèque de classes importer.

La ligne de code suivante correspond à la définition de la classe. Ici, il doit s'agir d'une classe publique, ce qui signifie qu'il est possible d'y accéder depuis l'animation principale. Le nom de la classe sera `HelloWorld3`, ce qui doit correspondre au nom du fichier, qui est **HelloWorld3.as**.

Cette classe étend `MovieClip`, ce qui signifie qu'elle fonctionnera avec un clip (dans le cas présent, la scène elle-même) :

```
public class HelloWorld3 extends MovieClip {
```

La classe contient une unique fonction, `HelloWorld3`, dont le nom correspond précisément à celui de la classe. Lorsqu'une fonction porte le même nom que la classe, elle est exécutée immédiatement quand la classe est initialisée. On appelle cette fonction la fonction constructeur.

Dans le cas présent, la classe est attachée à l'animation, de sorte qu'elle s'exécute dès que l'animation est initialisée. Dans la fonction se trouvent les trois mêmes lignes de code que nous avons utilisées dans l'exemple précédent :

```
        public function HelloWorld3() {
            var myText:TextField = new TextField();
            myText.text = "Hello World!";
            addChild(myText);
        }
    }
}
```

Pour que ce code fonctionne dans une animation, vous devez créer une nouvelle animation. L'exemple est appelé **HelloWorld3.fla**. Cette animation n'a rien besoin d'inclure dans le scénario, mais elle doit se voir associer une classe de document. Celle-ci indique le fichier ActionScript qui contrôle l'animation.

Pour définir une classe de document, ouvrez le panneau Propriétés en choisissant Fenêtre > Propriétés > Propriétés. Flash affiche le panneau présenté à la Figure 1.7. Ensuite, tapez le nom de classe `HelloWorld3` dans le champ Classe du document.

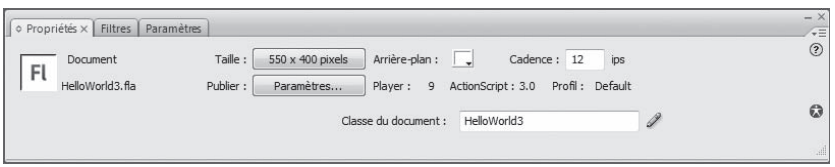


Figure 1.7
La classe du document de cette animation est HelloWorld3.

À présent, l'animation sait qu'elle doit charger et utiliser le fichier **HelloWorld3.as**. Lorsque vous la testez, le fichier de classe AS est compilé dans l'animation. L'exécution de l'animation initialise la classe, qui exécute la fonction `HelloWorld3` et affiche le texte "Hello World".

Travailler avec Flash CS3

Si l'essentiel de notre travail se fera en ActionScript, il est nécessaire de connaître certains termes et certaines notions concernant le travail avec le scénario, la scène et la bibliothèque de Flash CS3.

Si vous débutez avec Flash, tapez "Utiliser Flash" dans la documentation d'aide. Cette section livre une explication détaillée de la scène, du scénario, de la bibliothèque et d'autres éléments de l'espace de travail Flash et vous montre comment gérer l'interface Flash.

Objets d'affichage et listes d'affichage

Nous avons déjà traité des objets d'affichage. Il s'agit au fond de tous les éléments graphiques. Le plus polyvalent des objets d'affichage est le *clip*, un élément graphique complet qui peut inclure n'importe quel nombre d'autres objets d'affichage et possède un scénario pour l'animation.

Le *sprite* est une version plus simple du clip, qui ne contient qu'une seule image. Les objets d'affichage que nous créerons de toutes pièces en ActionScript seront généralement des sprites. Ils sont souvent plus efficaces que les clips parce qu'ils n'impliquent pas la surcharge requise pour inclure plusieurs images d'animation.

Parmi les autres objets d'affichage, on peut encore citer les champs texte, les images bitmap et les vidéos.

Certains objets d'affichage, comme les clips et les sprites, peuvent eux-mêmes contenir d'autres objets d'affichage. Par exemple, un sprite peut contenir plusieurs autres sprites, ainsi que des champs texte et des images bitmap.

L'imbrication des objets d'affichage offre un moyen d'organiser vos éléments graphiques. Par exemple, vous pouvez créer un unique sprite de jeu et y inclure tous les éléments du jeu que vous créez en ActionScript. Vous pouvez ensuite y insérer un sprite d'arrière-plan qui contient lui-même plusieurs sprites pour les éléments de l'arrière-plan. Un sprite contenant les pièces du jeu pourrait alors venir se placer au-dessus et servir de conteneur pour les pièces déplaçables.

Comme ils peuvent contenir plusieurs objets, les clips et les sprites conservent chacun une liste de ces éléments afin de déterminer l'ordre dans lequel ils doivent les afficher. On parle alors de *liste d'affichage*. Cette liste d'affichage peut être modifiée pour positionner des objets devant ou derrière d'autres objets.

Il est également possible de déplacer un objet d'affichage d'un objet parent à un autre. Cette opération ne copie pas l'objet : elle le supprime d'un côté et l'ajoute de l'autre. Ce système rend les objets d'affichage incroyablement flexibles et simples d'emploi.

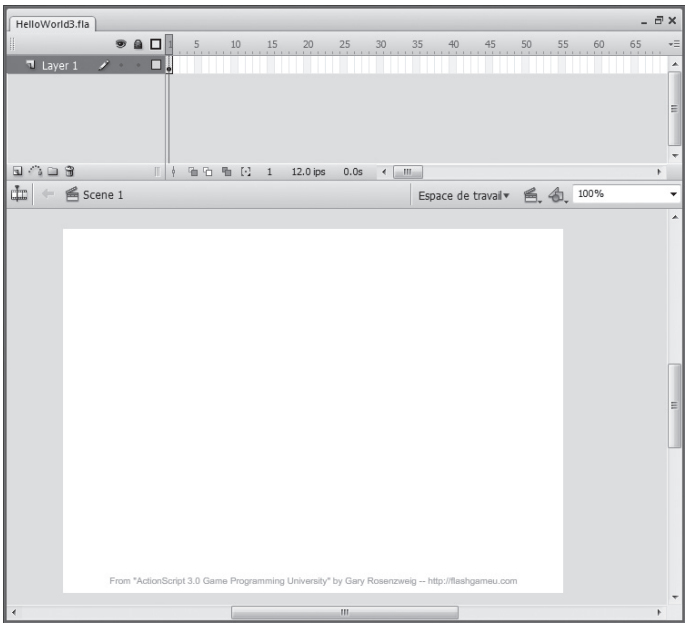
Le concept de liste d'affichage est une nouveauté d'ActionScript 3.0. Si vous avez l'habitude du mécanisme d'ActionScript 2.0, qui utilise des niveaux et des profondeurs, oubliez tout dès à présent et adoptez sans tarder la méthode plus simple des listes d'affichage. Grâce à ces listes, aucun objet ne se trouve à un niveau défini. L'objet d'affichage le plus lointain se trouve simplement être le premier dans la liste et le plus proche, le dernier. Vous pouvez à tout moment déplacer des objets dans la liste, et les risques d'erreur et les effets indirects sont considérablement réduits.

La scène

La scène est la principale zone de travail graphique dans Flash. Elle représente l'écran qui sera observé par les utilisateurs lorsqu'ils joueront à votre jeu.

La Figure 1.8 présente la fenêtre Document, dont la scène occupe la plus grande partie. Le scénario apparaît également en haut de la fenêtre.

Figure 1.8
La fenêtre Document inclut à la fois la scène et le scénario.



Un grand nombre de nos jeux posséderont une scène et un scénario complètement vides. Tous les éléments graphiques seront créés par le code ActionScript.

Dans le cas de quelques autres, des éléments graphiques se trouveront déjà dans la scène. Cette approche est particulièrement utile lorsqu'un concepteur graphique non programmeur s'attelle à créer un jeu. Il peut souhaiter disposer les éléments d'interface et les ajouter en cours de développement. Dans ce genre de cas, il n'est pas pratique que ces éléments soient créés par le code ActionScript.

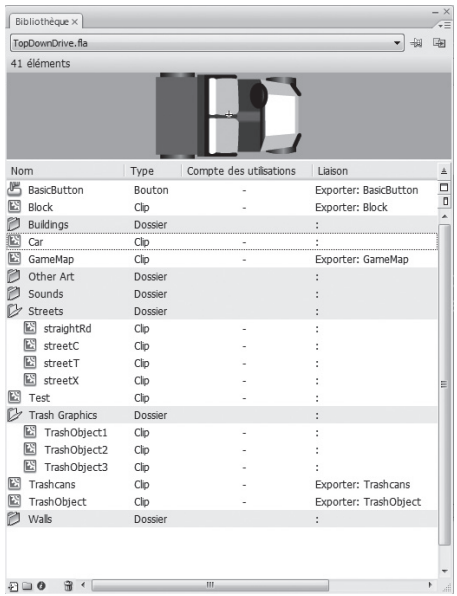
Au cours du développement, la scène peut être utilisée pour créer des éléments graphiques rapides. Par exemple, vous pouvez dessiner avec les outils de dessin dans la scène, sélectionner la forme, puis appuyer sur F8 pour créer un clip dans la bibliothèque.

La bibliothèque

La bibliothèque Flash contient tous les éléments multimédias requis dans votre jeu et se trouve intégrée dans le fichier SWF final. Vous pouvez importer d'autres éléments multimédias dans votre animation, comme vous le verrez lorsque nous importerons des images bitmap externes, au Chapitre 6.

La Figure 1.9 présente le panneau Bibliothèque. Il est généralement plus étroit, mais nous l'avons étiré pour faire apparaître la colonne Liaison.

Figure 1.9
Le panneau Bibliothèque présente tous les objets multimédias inclus dans l'animation.



À la Figure 1.9, la plupart des éléments de la bibliothèque sont des clips. Le premier est un bouton et quelques autres situés dans le dossier Sounds sont des sons.

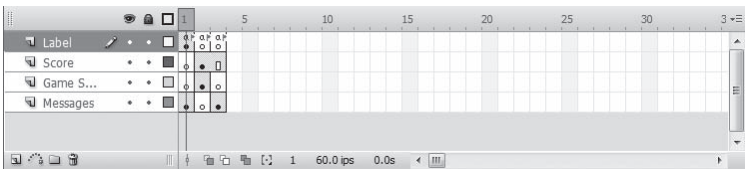
Certains des clips possèdent un nom d'exportation dans la colonne Liaison. Il s'agit d'éléments qui peuvent être extraits de la bibliothèque par notre code ActionScript à l'exécution.

Le scénario

L'animation Flash est décomposée en images. Le scénario en haut de la fenêtre permet de choisir l'image affichée dans la scène en bas de la fenêtre. Comme nous produisons non pas des animations mais des applications de jeu, nous allons utiliser les images pour différencier les écrans de nos jeux.

La Figure 1.10 présente un scénario. Seules trois images sont utilisées. Toutes sont des images-clés. La première est destinée à un écran d'introduction du jeu et contient des instructions. La seconde correspond à l'image dans laquelle la partie se joue. La troisième correspond à un message "Game Over" (partie terminée) et à un bouton "Play Again" (rejouer).

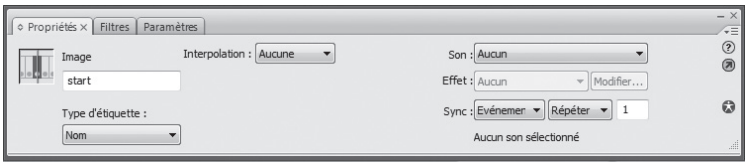
Figure 1.10
Le scénario a été légèrement étendu en utilisant le menu déroulant à droite afin que les images soient un peu agrandies.



Chaque image-clé possède une étiquette, bien qu'on ne puisse la voir dans le scénario. Un petit drapeau apparaît dans le calque supérieur de chaque image, qui signale la présence d'une étiquette à cet endroit.

Pour voir et définir les étiquettes d'image, vous devez sélectionner l'image puis vérifier le panneau Propriétés. Celui-ci contient un champ Image. Dans le cas présent, il contient la valeur "start" et vous pouvez le modifier si vous le souhaitez (voir Figure 1.11).

Figure 1.11
Le panneau Propriétés permet de définir ou de modifier l'étiquette de l'image.



Si vous examinez la Figure 1.10, vous remarquerez que l'animation contient quatre calques. Le premier, Label, contient trois images-clés. Pour créer des images et des images-clés, vous devez utiliser la touche F5 pour ajouter une image au calque et F7 pour ajouter une image-clé parmi ces images.

Le second calque, nommé "Score", ne contient que deux images-clés, l'image 1 et l'image 2. L'image 3 est simplement une extension de l'image 2. Les éléments de score présents durant le jeu à l'image 2 seront ainsi toujours présents à l'image 3.

Le scénario, la scène et la bibliothèque seront vos principaux outils visuels pour le développement de vos jeux.

Écrire et modifier du code ActionScript

S'il sera généralement nécessaire de travailler un peu dans le document Flash pour créer vos jeux, vous passerez le plus clair de votre temps dans la fenêtre Document ActionScript.

Vous avez déjà rencontré cette fenêtre à la Figure 1.6, mais la Figure 1.12 la présente sous un nouveau jour. Elle montre à gauche un menu hiérarchique de la syntaxe ActionScript 3.0.

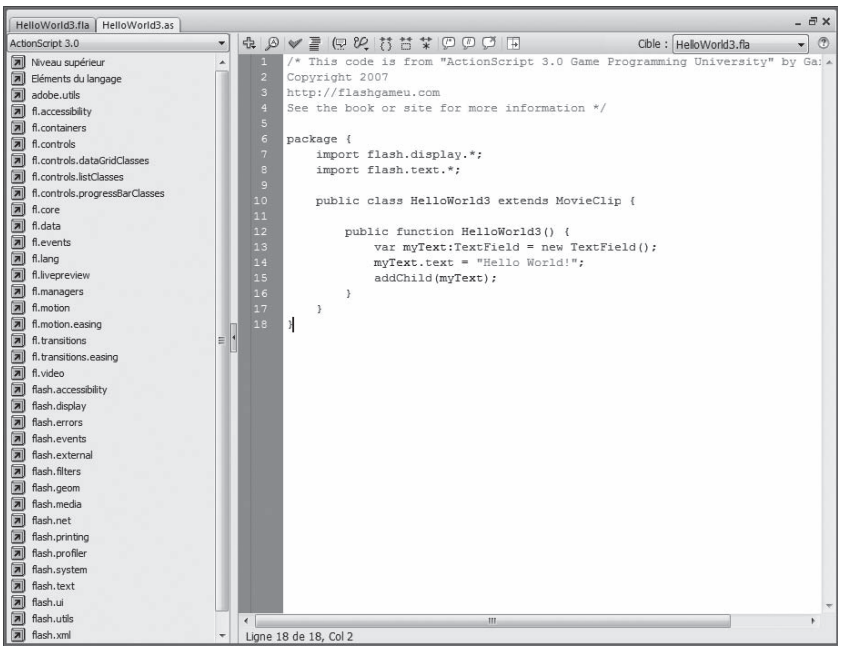


Figure 1.12
La fenêtre Document ActionScript contient plusieurs outils pratiques dans sa partie supérieure.

Tout en haut de la fenêtre apparaissent deux onglets. Deux documents sont en effet ouverts : **HelloWorld3.fla** et **HelloWorld3.as**. Ils permettent de travailler à la fois sur l'animation Flash et sur le document ActionScript. Vous pouvez passer de l'un à l'autre en cliquant sur les onglets. Vous pouvez également ouvrir d'autres fichiers ActionScript, ce qui peut se révéler pratique si vous travaillez avec plusieurs classes ActionScript à la fois.

Vous remarquerez à la Figure 1.12 que les lignes de code sont indentées. Le moyen approprié de créer ces retraits est d'utiliser la touche Tab. Lorsque vous appuyez sur Entrée à la fin d'une ligne de code, le curseur apparaît automatiquement en retrait au bon niveau de la ligne suivante. Si vous souhaitez supprimer une tabulation pour ramener une ligne vers la gauche, appuyez simplement sur Suppr ou sur Maj + Tab.

Vous pouvez également sélectionner une section de code et appuyer sur Tab pour déplacer tout le bloc vers la droite d'une tabulation. Utilisez Maj + Tab pour ramener le bloc d'une tabulation vers la gauche.

Les outils de la partie supérieure de la fenêtre de script réalisent différentes opérations et tout programmeur ActionScript se doit de savoir les utiliser. En voici une liste (comme le montre la fenêtre, de gauche à droite) :

- **Ajouter un nouvel élément au script.** Cet imposant menu déroulant offre accès à chacune des commandes ActionScript. Leur nombre est tel qu'il s'avère difficile à utiliser pour les commandes standard, mais il peut être utile pour les commandes plus rares.
- **Rechercher.** Utilisez cet outil pour ouvrir la boîte de dialogue Rechercher et remplacer. Vous pouvez également utiliser le raccourci Pomme + F (Mac) ou Ctrl + F (Windows).
- **Vérifier la syntaxe.** Cet outil permet d'amener le compilateur Flash à opérer une vérification de la syntaxe de votre script. Le résultat s'affiche dans le panneau Sortie.
- **Format automatique.** Cet outil reprend votre script entier et le reformate avec des tabulations, des espacements et des accolades cohérents. Si vous décidez de l'utiliser, assurez-vous de consulter la section Format automatique des Préférences afin d'opérer des décisions concernant les actions que ce bouton doit ou ne doit pas réaliser.
- **Afficher les conseils de code.** Voilà probablement le plus utile de tous ces boutons. Lorsque vous commencez à taper une fonction, par exemple `gotoAndStop(`, un conseil de code apparaît instantanément pour vous faire connaître les paramètres que prend la fonction. Si vous souhaitez éditer l'appel de fonction par la suite, vous pouvez positionner le curseur à l'intérieur des paramètres de la fonction et utiliser ce bouton pour afficher de nouveau les conseils.
- **Options de débogage.** Ce menu déroulant permet de définir et de supprimer des points d'arrêt. Nous traiterons du débogage par la suite, dans la section "Test et débogage" de ce chapitre.
- **Réduire entre les accolades.** Si vous cliquez sur ce bouton, la section courante de code entre accolades se réduit à une unique ligne. Le code reste présent, mais il est masqué. Vous pouvez cliquer sur le triangle (Mac) ou le signe plus (Windows) à gauche de la fenêtre ou sur le bouton Développer tout pour l'étendre. La Figure 1.13 montre à quoi ressemble le code lorsqu'une portion est réduite.
- **Réduire la sélection.** Ce bouton réduit le code actuellement sélectionné.
- **Développer tout.** Ce bouton ramène toutes les sections réduites à leur état normal.
- **Appliquer un commentaire de bloc.** Sélectionnez une portion de code et appuyez sur ce bouton pour transformer la sélection en un commentaire en la faisant précéder de `/*` et suivre de `*/`. Consultez la section "Stratégies de programmation des jeux ActionScript" pour en apprendre plus sur les commentaires dans le code.

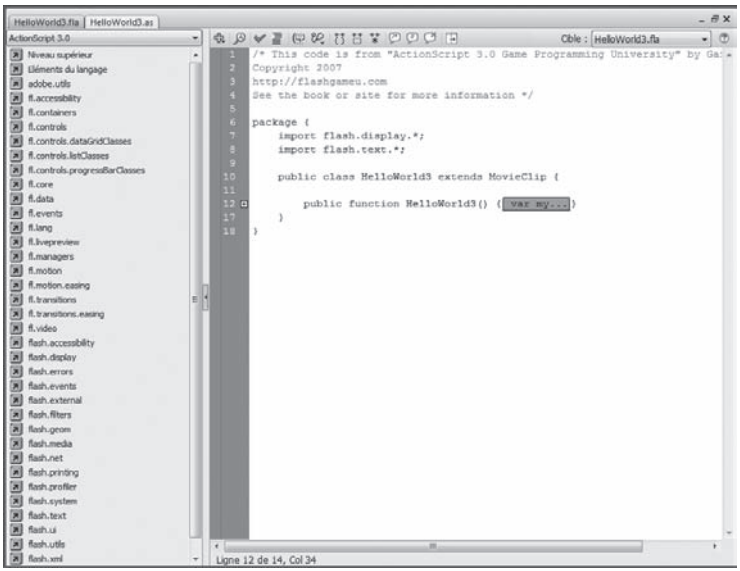


Figure 1.13
Un bloc de code a été réduit. Ce mécanisme est pratique lorsque vous travaillez sur un script très long et que vous souhaitez masquer les sections du code sur lesquelles vous ne travaillez pas pour le moment.

- **Appliquer un commentaire de ligne.** Cliquez sur ce bouton pour transformer la ligne courante en un commentaire. Si plusieurs lignes sont sélectionnées, toutes sont transformées en commentaire en ajoutant `//` au début de chacune.
- **Supprimer le commentaire.** Reconvertit les commentaires sélectionnés en code. Ce bouton est pratique lorsque vous souhaitez supprimer temporairement du code de votre programme. Vous pouvez transformer les lignes en commentaire afin qu'elles ne soient pas compilées puis supprimer les marques de commentaire pour faire réapparaître le code.
- **Afficher ou masquer la boîte à outils.** Ce bouton affiche ou masque la liste des commandes ActionScript à gauche de la fenêtre.

À droite des boutons se trouve un menu déroulant intitulé Cible qui permet de sélectionner le document d'animation Flash qui doit se compiler et s'exécuter lorsque vous sélectionnerez Contrôle > Tester l'animation. Il vous permet ainsi d'apporter une modification à votre code puis de tester l'animation sans avoir à revenir préalablement à la fenêtre Document. En général, c'est le dernier document d'animation Flash visualisé qui apparaît dans la liste, mais vous pouvez sélectionner un document particulier lorsque plusieurs documents sont ouverts.

Parmi les autres fonctionnalités importantes de la fenêtre Document ActionScript, il faut encore citer les numéros de ligne qui apparaissent côté gauche. Chaque ligne possède son propre numéro. Lorsque vous obtenez des erreurs de compilation en essayant de publier votre animation, elles font référence au numéro de ligne pour vous permettre de retrouver la source du problème.

Stratégies de programmation des jeux ActionScript

ActionScript 3.0 est très polyvalent. Vous pouvez adopter toutes sortes de styles de programmation et créer dans tous les cas des jeux qui fonctionnent bien.

Certains programmeurs préfèrent cependant certains styles plutôt que d'autres. Dans ce livre, j'ai adopté une méthode qui permet de se concentrer sur le code central du jeu, au détriment éventuel d'une organisation plus élaborée.

Méthode à une seule classe

Le troisième programme Hello World de ce chapitre est un simple fichier de classe lié à une animation Flash du même nom. Cette approche simple est rapide et facile.

L'autre option consiste à utiliser différents fichiers de classe pour les différents objets et processus du jeu. Il peut cependant alors devenir difficile de se rappeler où se trouvent les différentes portions de code des petits jeux. Par exemple, si une balle entre en collision avec une raquette dans un jeu, la détection de collision va-t-elle se trouver dans la classe de l'objet balle ou dans celle de l'objet raquette ?
Vous pouvez parfaitement décomposer le code en plusieurs classes si vous êtes familiarisé avec ce type d'organisation dans d'autres langages de programmation.

Avec un fichier de classe, toutes les propriétés de classe peuvent être clairement définies comme des variables en haut de la classe.

La classe du document contrôle le scénario principal, ce qui signifie qu'il est possible d'appeler des fonctions publiques dans la classe à partir de boutons placés dans la scène par les concepteurs. On peut aussi aisément contrôler le scénario principal en passant d'une image à une autre.

La méthode des petits pas

Voilà l'information qui risque bien d'être la plus importante du livre : si vous ne savez pas comment programmer quelque chose, décomposez votre problème en étapes plus petites et poursuivez ainsi jusqu'à ce que vous puissiez trouver la solution.

Les programmeurs débutants et certains programmeurs expérimentés qui oublient tout simplement cette règle se retrouvent souvent bloqués lorsqu'ils écrivent du code. Ils se disent : "Je ne sais pas comment faire pour que le programme réalise cette tâche." Le problème tient cependant uniquement au fait que la tâche n'en est pas une, mais en réalité plusieurs.

Par exemple, supposons qu'un programmeur souhaite faire pivoter un vaisseau spatial lorsque le joueur appuie sur les touches fléchées du clavier. Il se fruste parce qu'il ne sait pas comment réaliser cette tâche.

La solution consiste à décomposer la tâche de "rotation du vaisseau" : vérifier si la touche fléchée de gauche a été enfoncée ; soustraire un montant à la propriété `rotation` du sprite du vaisseau ; vérifier si la touche fléchée de droite a été enfoncée ; ajouter un montant à la propriété `rotation` du vaisseau.

La tâche de rotation du vaisseau correspond donc finalement à quatre petites tâches combinées en une.

Parfois, les programmeurs débutants font la même erreur à plus grande échelle. Ils supposent qu'ils ne pourront pas créer un jeu entier, parce qu'il semble trop complexe. Pourtant, si vous décomposez le jeu en une série de tâches de plus en plus petites (et vous occupez de ces tâches une par une), vous pourrez créer n'importe quelle sorte de jeu.

Un jeu de marteau simple pourra ne requérir qu'une centaine de tâches, alors qu'un jeu de plate-forme complexe en nécessitera plusieurs centaines. Chacune des tâches, si elle se trouve décomposée en ses étapes les plus élémentaires, se révélera cependant tout aussi facile à programmer.

Bonnes pratiques de programmation

Pendant que vous apprenez à utiliser le langage ActionScript 3.0 pour créer des jeux, il est aussi judicieux que vous gardiez à l'esprit quelques-unes des bonnes pratiques en matière de programmation. Il n'existe pas tant de règles que cela à respecter. Je m'autorise moi-même à les enfreindre à certaines occasions dans ce livre. Il ne fait pourtant aucun doute que vous serez meilleur programmeur si vous commencez par bien assimiler ces bonnes pratiques.

Du bon usage des commentaires

Ponctuez votre code de commentaires simples mais signifiants.

Ce petit effort supplémentaire du moment vous attirera tous les éloges quelques mois plus tard lorsque vous devrez vous replonger dans vos programmes pour les modifier.

Si vous travaillez avec d'autres programmeurs ou si vous pensez qu'il y a la moindre chance qu'une autre personne puisse avoir à modifier votre code à l'avenir, ce simple conseil doit faire figure de règle impérative.

Il existe généralement deux types de commentaires : les commentaires de ligne et les commentaires de bloc. Le commentaire de ligne est une simple remarque en style télégraphique à la fin d'une ligne ou quelquefois une simple ligne de commentaire avant la ligne de code. Le commentaire de bloc est

un commentaire plus important, généralement d'une ou de plusieurs phrases, qui apparaît avant une fonction ou une section de code :

```
someActionScriptCode(); // Voici un commentaire de ligne

// Voici un commentaire de ligne
someActionScriptCode();

/* Voici un commentaire de bloc.
Les commentaires de bloc sont plus longs.
Et contiennent une description concernant le code qui suit. */
```

Il est également important de rendre vos commentaires signifiants et concis. Ne vous contentez pas de reformuler ce que le code évoque en lui-même, comme ceci :

```
// Boucle 10 fois
for (var i:int=0;i<10;i++) {
```

En outre, n'utilisez pas de paragraphe de texte lorsque quelques mots suffisent. Les commentaires inutilement longs peuvent devenir aussi inefficaces que s'ils étaient absents. Ne cherchez pas à en faire trop.

Utiliser des noms de variable et de fonction descriptifs

Ne craignez pas d'utiliser des noms longs et descriptifs pour vos variables et vos fonctions. Vous créerez ainsi du code qui s'explique en partie de lui-même. Prenez cet exemple :

```
public function placerTruc() {
    for(var i:int=0;i<10;i++) {
        var a:Chose = new Chose();
        a.x = i*10;
        a.y = 300;
        addChild(a);
    }
}
```

Que fait ce code ? Il semble placer des copies d'un clip à l'écran. Mais quel clip, dans quel but ? Voici une autre version du programme :

```
public function placerPersonnagesEnnemis() {
    for(var numEnnemi:int=0; numEnnemi<10; numEnnemi++) {
        var ennemi:PersonnageEnnemi = new PersonnageEnnemi();
        ennemi.x = numEnnemi*10;
        ennemi.y = 300;
        addChild(ennemi);
    }
}
```

Il sera déjà bien plus facile de revenir à ce code quelques mois plus tard.

L'exception courante à cette règle concerne l'utilisation de la variable `i` comme variable incrémentale dans les boucles. Dans l'exemple précédent, j'aurais normalement conservé la variable `i` au lieu de la remplacer par `numEnnemi`. Les deux approches conviennent, mais il est devenu assez standard pour les programmeurs d'utiliser la variable `i` dans les boucles. En fait, les programmeurs poursuivent généralement cette logique en utilisant les variables `j` et `k` dans les boucles `for` imbriquées.

Transformer le code répétitif ou similaire en fonction

Si vous devez utiliser la même ligne de code plusieurs fois dans un programme, envisagez de la transformer en une fonction et d'appeler cette fonction à la place.

Par exemple, il se peut que vous souhaitiez mettre à jour le score à plusieurs endroits de votre jeu. Si le score est affiché dans un champ texte nommé `scoreDisplay`, vous procéderez ainsi :

```
scoreDisplay.text = "Score: "+playerScore;
```

Au lieu d'inclure cette même ligne de code à cinq endroits différents, il est préférable de placer un appel de fonction dans ces cinq emplacements :

```
showScore();
```

Ensuite, la fonction peut prendre la forme suivante :

```
public function showScore() {
    scoreDisplay.text = "Score: "+playerScore;
}
```

À présent que ce code est situé à un seul endroit, il devient très facile de changer le terme `Score` et de le remplacer par `Points`. Inutile d'effectuer une opération de recherche et de remplacement dans le code : il n'y a qu'un seul endroit à modifier.

Vous pouvez en faire de même lorsque le code n'est pas identique. Par exemple, supposons que vous ayez une boucle dans laquelle vous placez dix copies du clip A à gauche de la scène et une autre dans laquelle vous placez dix copies du clip B à droite de la scène. Vous pouvez créer une fonction qui prend la référence du clip et la position horizontale de l'emplacement et positionne les clips en fonction de ces paramètres. Ensuite, vous pouvez appeler votre fonction deux fois, une fois pour le clip A et une autre pour le clip B.

Tester votre code par petits blocs

À mesure que vous écrivez votre code, testez-le sur des portions aussi réduites que possible. Vous pourrez ainsi capturer les erreurs à mesure que vous écrivez votre code.

Par exemple, si vous souhaitez créer une boucle qui place au hasard dix cercles de couleur différente à l'écran, vous devez d'abord créer les dix cercles en les positionnant de manière aléatoire. Testez ce code et faites-le fonctionner comme vous le souhaitez. Ensuite, ajoutez la fonctionnalité qui définit aléatoirement la couleur.

Il s'agit en réalité d'une variante de la "méthode des petits pas". Décomposez votre tâche de programmation en petites étapes. Créez le code pour chaque étape et effectuez un test à chaque fois.

Notions élémentaires du langage ActionScript

Examinons la syntaxe de programmation la plus élémentaire en ActionScript 3.0. Si vous débutez avec ActionScript mais que vous ayez déjà utilisé un autre langage de programmation, vous pourrez ainsi découvrir rapidement le fonctionnement d'ActionScript.

Si vous avez déjà utilisé ActionScript ou ActionScript 2.0, vous aurez l'occasion de constater les différences introduites avec ActionScript 3.0.

Créer et utiliser des variables

Le stockage de valeurs peut s'opérer en ActionScript 3.0 à l'aide d'une simple instruction d'attribution. Vous devez cependant déclarer vos variables la première fois que vous les utilisez. Pour cela, vous pouvez placer le mot-clé `var` avant le premier usage de la variable :

```
var maValeur = 3;
```

Vous pouvez aussi commencer par déclarer votre variable et l'utiliser par la suite :

```
var maValeur;
```

L'instruction `var` n'est pas requise avec ActionScript 2.0. Elle l'est en revanche avec ActionScript 3.0.

Lorsque vous créez une variable de cette manière, elle prend le type polyvalent `Object`. Cela signifie qu'elle peut contenir n'importe quel type de valeur de variable : un nombre, une chaîne comme "Hello" ou un contenu plus complexe comme un tableau ou une référence de clip.

Si vous déclarez une variable comme étant d'un type spécifique, vous ne pourrez en revanche l'utiliser que pour stocker des valeurs du même type :

```
var maValeur:int = 7;
```

Une variable de type `int` peut être un entier, positif ou négatif. Les variables de type `uint` sont destinées uniquement aux entiers positifs. Si vous souhaitez utiliser des valeurs fractionnelles (aussi appelées nombres à virgule flottante), vous devez utiliser le type `Number` :

```
var maValeur:Number = 7.8;
```

Vous pouvez aussi utiliser les types `String` et `Boolean`. Les valeurs `String` contiennent du texte, tandis que les valeurs `Boolean` correspondent à `true` ou à `false`.

Nous venons d'énumérer les types primitifs de base. Vous pouvez cependant avoir aussi des tableaux, des références de clip et de sprite et de nouveaux types qui correspondent aux classes de code que vous créez.

Il y a un avantage évident en termes d'efficacité à utiliser des variables précisément définies. Par exemple, l'accès aux valeurs `int` se fait bien plus rapidement qu'aux valeurs `Number`. Vous pouvez ainsi contribuer à accélérer des processus critiques dans vos jeux en choisissant des types aussi basiques que possible pour toutes vos variables.

Les opérations sur les variables numériques ressemblent à celles de presque tous les autres langages de programmation. L'addition, la soustraction, la multiplication et la division s'effectuent avec les opérateurs `+`, `-`, `*` et `/` :

```
var monNombre:Number = 7.8+2;
var monAutreNombre:int = 5-6;
var monAutreNombre:Number = monNombre*3;
var monNombreSuivant:Number = monNombre/monAutreNombre;
```

Vous pouvez également utiliser des opérateurs spéciaux pour simplifier les opérations. Par exemple, l'opérateur `++` incrémente d'une unité la variable à laquelle il s'applique. L'opérateur `--` lui soustrait une unité :

```
monNombre++;
```

Vous pouvez utiliser `+=`, `-=`, `*=` et `/=` pour réaliser une opération sur la variable d'origine. Par exemple, l'instruction suivante ajoute sept à la variable :

```
monNombre += 7;
```

Vous pouvez également utiliser des parenthèses pour définir l'ordre des opérations :

```
var monNombre:Number = (3+7)*2;
```

Les valeurs `String` peuvent également être manipulées avec l'opérateur `+` et l'opérateur `+=` :

```
var maValeurString:String = "Hello";
var monAutreValeurString = maValeurString+"World";
maValeurString += "World";
```

Lorsque nous utilisons des variables dans des classes, elles deviennent des propriétés de cette classe. Dans ce cas, il faut encore les définir comme privées ou publiques. Les variables privées ne sont pas accessibles depuis le code hors de la classe. Dans la plupart des cas, cela ne pose pas de problème, car les fonctions de classe doivent être les seules à pouvoir modifier les valeurs des variables de classe.

Instructions conditionnelles

L'instruction `if` fonctionne en ActionScript comme elle le fait dans de nombreux langages de programmation :

```
if (maValeur == 1) {
    faireQuelqueChose();
}
```

L'opérateur de comparaison == vérifie l'égalité générale. Vous pouvez également utiliser >, <, >= et <= pour déterminer si une valeur est supérieure, inférieure, supérieure ou égale ou inférieure ou égale à une autre.

Il est aussi possible d'ajouter les mots-clés `else` et `else if` pour étendre la structure `if` :

```
if (maValeur == 1) {
    faireQuelqueChose();
} else if (maValeur == 2) {
    faireAutreChose();
} else {
    neRienFaire();
}
```

Vous pouvez également inclure des conditions plus complexes avec && et | |. Ces signes représentent les opérateurs de comparaison ET et OU.

Avant ActionScript 3.0, il était possible d'utiliser les mots-clés `and` et `or` à la place de && et | |. Ces mots-clés ne sont maintenant plus acceptés.

```
if ((maValeur == 1) && (maChaine == "This")) {
    faireQuelqueChose();
}
```

Boucles

Les boucles s'effectuent avec l'instruction `for` ou l'instruction `while`.

L'instruction `for` contient trois parties : l'instruction initiale, une condition et une instruction de changement. Par exemple, le code suivant positionne la variable `i` à zéro, boucle tant qu'elle est inférieure à dix et augmente la valeur de `i` à chaque passage dans la boucle :

```
for(var i:int=0;i<10;i++) {
    faireQuelqueChose();
}
```

Vous pouvez utiliser la commande `break` pour quitter une boucle à tout moment. La commande `continue` ignore le reste des lignes de code dans la boucle et commence la prochaine itération dans la boucle.

La boucle `while` est une boucle qui se poursuit indéfiniment jusqu'à ce qu'une condition initiale soit remplie :

```
var i:int = 0;
while (i < 10) {
    i++;
}
```

La boucle `do` est une variante de la boucle `while`. Elle lui est pour l'essentiel identique, à ceci près que l'instruction conditionnelle intervient après la boucle, ce qui lui garantit de s'exécuter au moins une fois :

```
var i:int = 0;
do {
    i++;
} while (i <10);
```

Fonctions

Pour créer une fonction avec ActionScript 3.0, vous devez simplement déclarer la fonction, les paramètres qui lui sont passés en entrée et la sortie qu'elle retourne. Ensuite, vous définissez la fonction avec le code qu'elle contient.

Si la fonction se trouve dans une classe, vous devez également indiquer s'il s'agit d'une fonction publique ou d'une fonction privée. Les fonctions privées ne sont pas accessibles depuis l'extérieur de la classe. Avec notre méthode de développement de jeu à classe unique, nous utiliserons principalement des classes privées.

Vous constaterez parfois que les fonctions sont appelées des méthodes. Dans la documentation, le terme de méthode est fréquemment utilisé, mais c'est le mot-clé `function` *qui est adopté pour la définition, comme vous le verrez ci-après. Je préfère donc pour ma part utiliser le terme de fonction.*

Voici une fonction simple à l'intérieur d'une classe. Si cette fonction se trouvait dans le scénario principal au lieu d'être dans une classe, nous pourrions ignorer le mot-clé `private` :

```
private function maFonction(monNombre:Number, maChaine:String): Boolean {
    if (monNombre == 7) return true;
    if (maChaine.length < 3) return true;
return false;
}
```

Cette fonction d'exemple se contente de retourner `true` si le nombre vaut sept ou si la longueur de la chaîne est inférieure à trois caractères. Il s'agit d'un exemple simple qui illustre la syntaxe de création d'une fonction.

Test et débogage

Personne ne peut prétendre écrire du premier jet un code absolument parfait, pas même le programmeur le plus chevronné au monde. Il est donc nécessaire de programmer votre code, de le tester et de le déboguer.

Types de bogues

Il existe trois raisons de déboguer votre code. La première concerne le cas où vous obtenez un message d'erreur lors de la compilation ou de l'exécution de votre code. Vous devrez alors retrouver le problème et le corriger. En général, vous le découvrirez immédiatement. Il pourrait par exemple s'agir d'un nom de variable mal orthographié.

La seconde raison concerne le cas où le programme ne fonctionne pas comme prévu. Par exemple, un personnage censé bouger ne le fait pas, l'entrée de l'utilisateur n'est pas acceptée ou les balles tirées par le héros traversent l'ennemi sans le toucher. Ce type de bogue doit être analysé et traqué et cette opération peut parfois prendre du temps.

La troisième raison de déboguer votre code concerne le cas où vous souhaitez l'améliorer. Vous pouvez repérer les inefficacités et les problèmes qui génèrent des ralentissements. Parfois, ces problèmes sont aussi critiques que des bogues, car un jeu lent peut devenir aussi injouable qu'un jeu dysfonctionnant.

Le type de question le plus courant que j'obtiens de la part d'autres programmeurs concerne des cas où le code ne fonctionne pas comme prévu. Puis-je leur dire ce qui cloche ?

Oui, je le peux, mais la réponse se trouve juste devant eux. Il suffit qu'ils exploitent leurs compétences en débogage pour le savoir. Or, en qualité de créateur du code, ce sont eux les mieux placés pour le faire.

Méthodes pour le test

Il existe plusieurs manières de pister les problèmes dans votre code. L'approche la plus simple consiste tout simplement à le reparcourir mentalement. Par exemple, parcourez le code suivant ligne par ligne et effectuez les calculs par vous-même, comme si vous étiez l'ordinateur :

```
var monNombre:int = 7;
monNombre += 3;
monNombre *= 2;
monNombre++;
```

Inutile d'exécuter ce code pour savoir que la valeur de `monNombre` est à présent 21.

Dans les cas où le code est trop long ou lorsque les calculs sont trop difficiles à réaliser, une simple commande `trace` transmet des informations au panneau Sortie afin que vous puissiez les examiner :

```
var monNombre:int = 7;
monNombre += 3;
monNombre *= 2;
monNombre++;
trace("monNombre = ", monNombre);
```

Avant Flash CS3, la commande `trace` n'acceptait qu'une unique chaîne et la transmettait au panneau Sortie. Elle peut maintenant accepter n'importe quel type de données, ainsi que des éléments multiples séparés par des virgules. Ces nouvelles capacités en font maintenant un outil bien plus utile pour le débogage.

J'utilise moi-même assez souvent des instructions `trace` en cours de développement. Par exemple, si le joueur doit opérer une série de choix au début du jeu, j'envoie les résultats de ces choix vers le panneau Sortie avec `trace`. J'obtiens ainsi pendant que j'effectue mes tests un rappel des options que j'ai choisies avant de jouer au jeu si quelque chose d'inattendu se produit.

Utiliser le débogueur

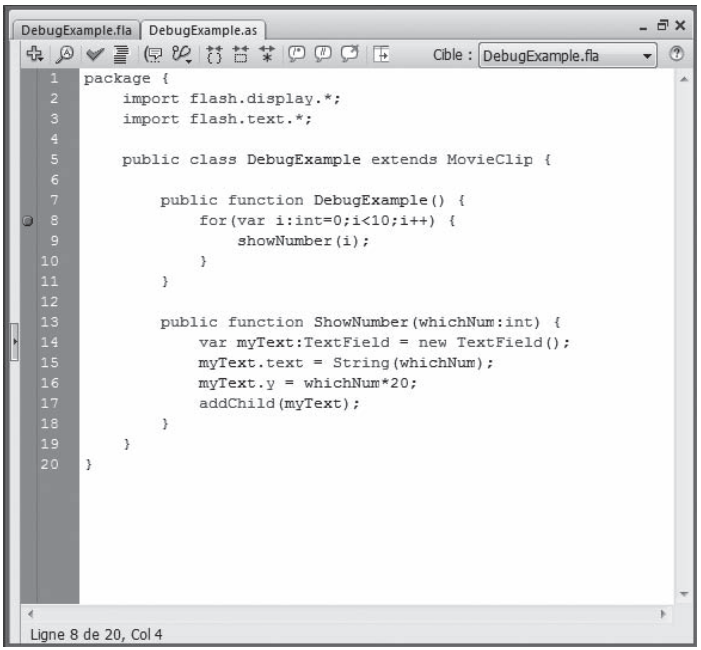
Flash CS3 vous permet d'utiliser un débogueur d'exécution pour tester votre code pendant que votre animation s'exécute. Si vous utilisiez auparavant le débogueur ActionScript 2.0, ne vous fiez pas à ces anciennes sensations : le débogueur ActionScript 3.0 est un tout autre animal.

Définir un point d'arrêt

Le moyen le plus simple de déboguer un programme consiste à définir un point d'arrêt. Vous pouvez le faire en sélectionnant une ligne de votre code et en choisissant Déboguer > Basculer le point d'arrêt dans la barre des menus. Vous pouvez également appuyer sur Commande + B (Mac) ou Ctrl + B (Windows) pour définir ou supprimer un point d'arrêt.

La Figure 1.14 présente le code **DebugExample.as**, dans lequel un point d'arrêt est défini. Il apparaît sous la forme d'un cercle rouge à gauche de la fenêtre avant la huitième ligne. Le programme crée simplement dix champs texte contenant les numéros 0 à 9 et les place verticalement du côté gauche de l'écran.

Figure 1.14
Le curseur a été placé à la ligne 8 avant de choisir Déboguer >Basculer le point d'arrêt afin de définir un point d'arrêt à cet endroit.



Une fois qu'un point d'arrêt est défini, vous pouvez utiliser la commande Déboguer > Déboguer l'animation au lieu de Contrôle > Tester l'animation pour tester votre animation. Lorsque le programme atteint la ligne où se trouve le point d'arrêt, il s'interrompt et affiche une variété d'informations dans différentes fenêtres de débogage.

Si vous utilisez la commande Déboguer > Déboguer l'animation avec le point d'arrêt à la ligne 8, toute une série de panneaux s'affiche en plus de l'animation Flash qui s'exécute (voir Figure 1.15).

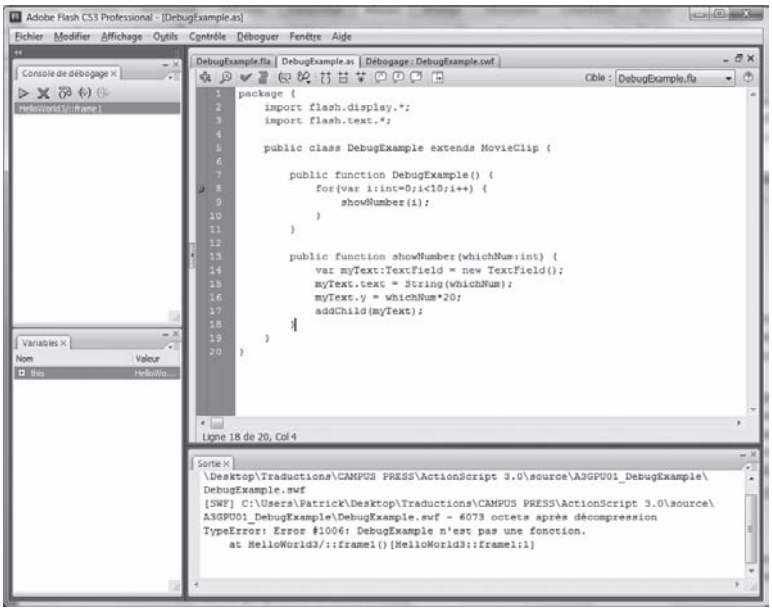


Figure 1.15
Le panneau Débogage présente diverses informations concernant l'état de votre programme.

Parcourir le code au pas à pas

Cinq boutons apparaissent en haut de votre panneau Console de débogage dans le coin supérieur gauche. Le premier est le bouton Continuer, qui reprend la lecture de l'animation à partir du point où elle s'est interrompue. Le second est un X. Il interrompt la session de débogage et relance l'animation à partir de ce point sans débogage.

Les trois autres boutons concernent le parcours du code au pas à pas. Le premier, Pas à pas principal, exécute la ligne courante et passe à la suivante. Si la ligne de code courante appelle une autre fonction, il exécute cette fonction. Le bouton suivant, Pas à pas détaillé, introduit le programme dans une nouvelle fonction s'il en existe une dans la même ligne. En utilisant ce bouton de manière répétitive, vous pourrez consulter une à une chaque ligne individuelle du programme au lieu de sauter les appels de fonction.

Le dernier bouton ressort de la fonction courante. Utilisez-le pour terminer l'exécution de la fonction courante et passer à la ligne suivante de la fonction que vous venez de quitter.

La Figure 1.16 présente les panneaux débogage une fois que vous êtes entré dans la fonction `showNumber` puis descendu de quelques lignes. Comme vous pouvez le voir, le panneau Variables présente la valeur de `i`. Vous pouvez également étendre la variable `myText` pour voir toutes les propriétés du champ texte.

En haut à gauche, le panneau de débogage indique où vous vous trouvez dans le programme. Dans cet exemple, vous vous trouvez actuellement dans la fonction `showNumber`, qui a été appelée à partir de la fonction constructeur de cette classe. Cette indication se révèle pratique lorsque vous avez une fonction qui peut être appelée à partir de plusieurs emplacements.

Le fait de savoir comment utiliser le débogueur pour corriger des bogues et des comportements inattendus est aussi important que savoir comment écrire du code. Lorsque vous travaillerez avec les jeux de ce livre et tenterez de les modifier pour les adapter à vos besoins, apprenez également à déboguer votre code.

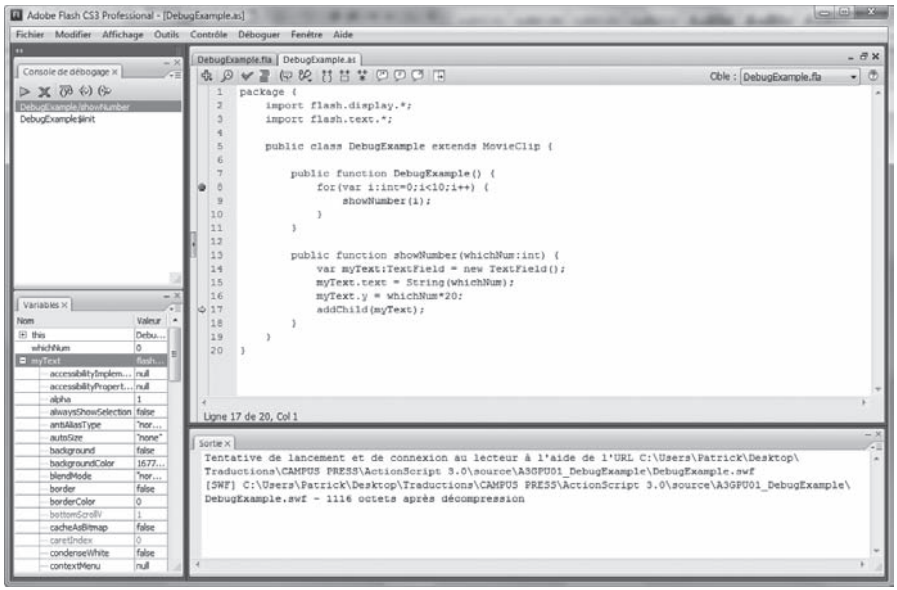


Figure 1.16
Les panneaux de débogage présentent l'état du programme à mesure que vous en parcourez les différentes étapes.

Publier votre jeu

Une fois que vous aurez terminé un jeu, que vous l'aurez testé et aurez été satisfait, il sera temps de le publier. Les jeux Flash se publient généralement sur le Web en étant incorporés dans des pages Web.

Flash CS3 facilite considérablement ce processus, mais il convient de comprendre quelques options avant de se lancer dans la publication.

Pour accéder à la boîte de dialogue Paramètres de publication, choisissez Fichier >Paramètres de publication. La boîte de dialogue Paramètres de publication contient trois sections : Formats, Flash et HTML.

Formats

Les paramètres Formats (voir Figure 1.17) permettent de sélectionner les fichiers à exporter.

Les formats d'image désignent des substitutions lorsque l'utilisateur ne possède pas de lecteur Flash. L'exportation QuickTime sert à placer une animation Flash 5 dans un fichier QuickTime. Aucun de ces paramètres ne nous concerne en tant que développeurs de jeux ActionScript 3.0.

Les deux options Projection peuvent être utilisées pour créer des versions autonomes de nos jeux. Elles représentent un moyen complètement différent d'exporter vos animations terminées.

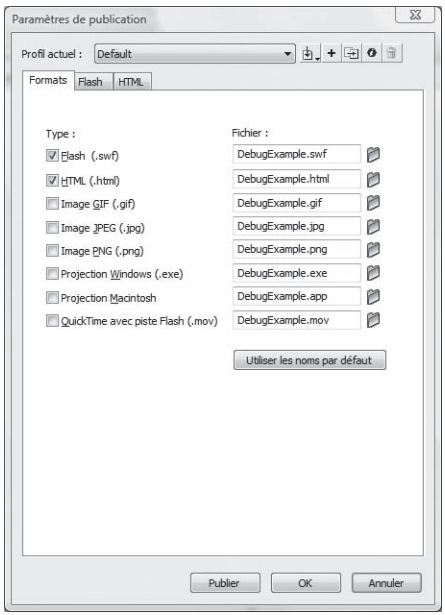


Figure 1.17
Seuls les formats Flash et HTML sont sélectionnés pour l'exportation.

Si vous possédez déjà un modèle de page Web personnalisé que vous utilisez sur votre site, l'option HTML ne sera pas forcément nécessaire. Dans ce cas, vous n'avez pas besoin de page par défaut dans laquelle incorporer votre jeu. Vous pouvez cependant l'exporter quoi qu'il en soit puis récupérer le code du corps de la page d'exemple afin de l'utiliser à l'emplacement approprié de votre page personnalisée.

Flash

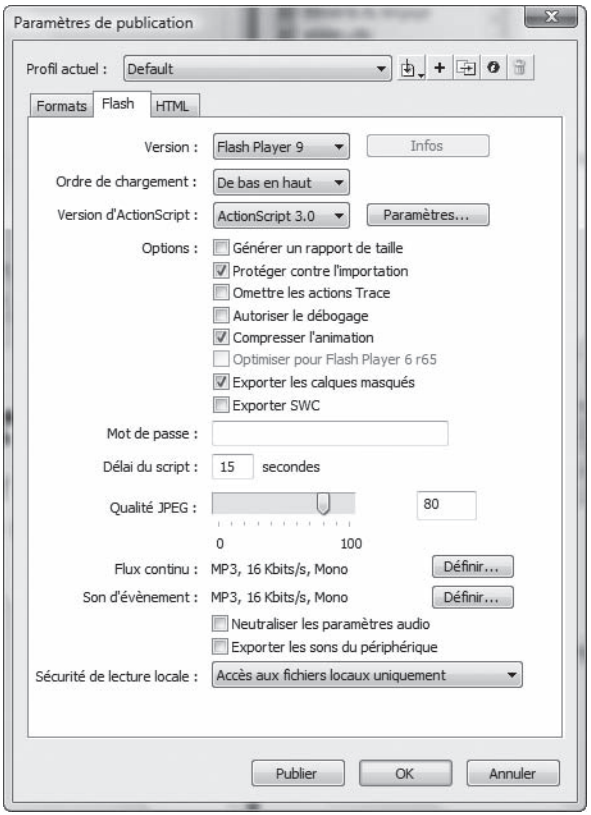
Les paramètres Flash sont les plus importants pour l'exportation d'une animation Flash complexe comme nos jeux. De préférence, exportez des fichiers Flash Player 9, en choisissant la version ActionScript 3.0 (voir Figure 1.18).

Cochez également l'option Protéger contre l'importation. Il deviendra ainsi plus difficile pour vos utilisateurs de télécharger votre animation et de la modifier pour se l'approprier.

Malheureusement, il n'existe pas de moyen garanti de protéger votre animation Flash après qu'elle a été publiée sur le Web. Il existe des programmes de décompilation qui permettent de récupérer un fichier SWF compressé et protégé et de le reconvertir en une animation FLA modifiable. Les options Protéger contre l'importation et Compresser l'animation compliquent ce processus mais ne protègent pas de manière absolue contre ces techniques.

Le reste des paramètres Flash concerne les paramètres de compression et de sécurité. Consultez la documentation Flash pour une documentation détaillée concernant chacun de ces réglages.

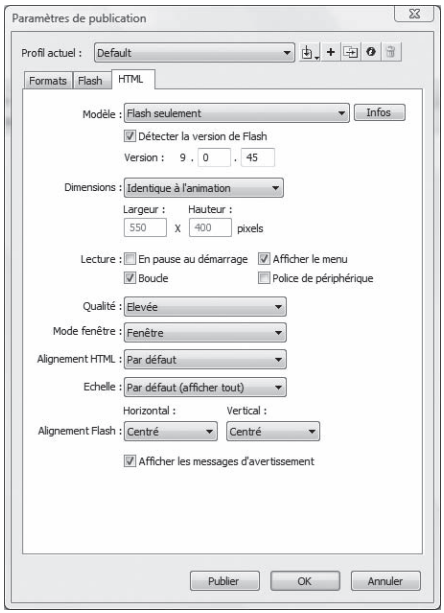
Figure 1.18
Ces paramètres sont conseillés pour l'utilisation habituelle des jeux Flash.



HTML

Les paramètres HTML n'importent que si vous souhaitez utiliser la page HTML créée avec la commande Publier. Il est cependant judicieux de voir comment Adobe considère qu'il est préférable de publier vos animations Flash. La Figure 1.19 présente ces options.

Figure 1.19
Les paramètres HTML vous permettent de choisir un modèle HTML à exporter avec l'animation Flash.



Le réglage par défaut (Flash uniquement) utilise du code JavaScript pour incorporer l'animation. Il utilise le fichier AC_RunActiveContent.js qui est également généré lors de la publication. La page HTML principale charge ensuite le JavaScript dans ce fichier, qui place l'animation Flash dans une balise `<div>` à l'intérieur de la page Web.

Pourquoi se donner la peine d'utiliser du JavaScript alors qu'il serait possible de n'utiliser qu'une simple balise `<object>`/`<embed>`, comme vous l'avez fait les années précédentes ? En raison d'un conflit portant sur des brevets, Microsoft a dû changer la manière d'incorporer les éléments multimédias dans les pages sous Internet Explorer. Tous les éléments multimédias incorporés directement dans une page requièrent maintenant un clic pour être activés dans Internet Explorer 7 et certaines versions d'Internet Explorer 6. Cette méthode JavaScript évite cependant ce clic supplémentaire.

L'une des options souvent utilisées consiste à amener l'animation Flash à se redimensionner pour remplir la fenêtre entière du navigateur. Cette mise à l'échelle peut s'effectuer en choisissant l'option Pourcentage dans le menu déroulant Dimensions et en fixant la Largeur et la Hauteur à 100. L'animation s'étend alors de manière à remplir la fenêtre tout en conservant ses proportions.

L'option Taille exacte de la liste déroulante Échelle lui permet de perdre ses proportions d'origine et de s'étendre verticalement pour s'adapter à la hauteur de la fenêtre et horizontalement pour couvrir sa largeur.

Comme tous les graphismes vectoriels se redimensionnent harmonieusement dans Flash et que votre code peut fonctionner sans problème à n'importe quelle échelle, il est généralement judicieux de permettre à l'utilisateur d'ajuster la taille du jeu en modifiant simplement la taille de sa fenêtre de navigateur. Les utilisateurs qui possèdent de petits écrans et les autres dotés de grands pourront ainsi jouer aux jeux comme ils l'entendent.

Le paramètre Qualité fait partie des autres options disponibles. Avec le réglage Élevée, le lecteur Flash reproduit l'image à haute résolution afin d'obtenir le meilleur effet de lissage sur les bords des formes vectorielles. L'option Moyenne réduit la résolution du lissage mais améliore les performances de l'animation. Avec l'option Élevée automatiquement, Flash tente d'utiliser le réglage Élevée mais repasse à l'option Moyenne si la lecture est trop lente. Le réglage Inférieure supprime le lissage mais propose la meilleure vitesse de lecture.

Chek-list de la programmation de jeux ActionScript

Lorsque vous créez un jeu Flash, plusieurs facteurs doivent être pris en compte. Un élément clé peut toujours être oublié qui amènerait le jeu à ne pas fonctionner correctement. Pour éviter certains problèmes simples, voici une check-list rapide à laquelle vous pourrez vous reporter à l'occasion.

Paramètres de publication et de document

Il est important de se rappeler qu'il existe des paramètres essentiels dans la boîte de dialogue Paramètres de publication et dans le panneau Propriétés de l'animation.

La classe du document est-elle correctement définie ?

La Figure 1.7 montre comment définir la classe du document en utilisant le panneau Propriétés de l'animation. Si vous oubliez de définir ce réglage, l'animation s'exécutera et ignorera tout simplement la classe que vous avez créée.

Les paramètres de publication sont-ils correctement définis ?

Assurez-vous de définir les Paramètres de publication de manière que l'animation Flash soit compilée pour Flash 9 et ActionScript 3.0. Il est assez improbable que votre animation se compile si ces paramètres ne sont pas correctement définis, mais cela reste possible.

Vérifier les paramètres de sécurité

Dans la section Flash des Paramètres de publication figure un paramètre Sécurité de lecture locale. Vous pouvez choisir l'option Accès aux fichiers locaux uniquement ou l'option Accès au réseau uniquement. Afin de vous assurer que vos animations Flash sont sécurisées, choisissez l'option appropriée.

Ce réglage peut poser problème si vous devez accéder à des fichiers locaux et que l'option Accès réseau uniquement soit sélectionnée. Si vous utilisez des fichiers externes et que quelque chose ne se passe pas comme prévu lorsque vous chargez les fichiers sur un serveur, commencez par vérifier ces paramètres.

Noms des classes, des fonctions et des variables

Même en vous efforçant de suivre les bonnes pratiques de programmation précédemment mentionnées, il se peut que vous commettiez des erreurs simples parfois difficiles à retrouver.

Penser à la casse des caractères

Lorsque vous nommez une variable ou une fonction, la casse des caractères est prise en compte. `maVariable` et `mavariable` sont ainsi complètement différentes. De la même manière, une classe nommée `maClasse` exécute la fonction `maClasse` lorsqu'elle s'initialise. Si vous avez nommé votre fonction `maclasse`, elle ne sera pas appelée.

Les incohérences de noms de variable sont habituellement capturées par le compilateur, car un nom de variable mal orthographié n'est pas initialisé, mais il reste possible d'oublier de déclarer une variable et de la déclarer de nouveau avec une autre casse de caractères. Voilà le genre d'erreur à surveiller.

Les fichiers de classe de l'animation sont-ils présents ?

Si un clip se voit attribuer des propriétés Liaison que le code ActionScript doit utiliser, il peut utiliser la classe dynamique par défaut ou vous pouvez créer une classe pour lui. Par exemple, vous pouvez créer un clip `EnemyCharacter` et lui associer un fichier de classe **EnemyCharacter.as**.

Rien de plus facile cependant que d'oublier cette classe ou de l'orthographier de manière incorrecte. Par exemple, un fichier **Enemycharacter.as** sera tout simplement ignoré et ne sera pas lié au clip `EnemyCharacter`.

Les classes étendent-elles le bon type ?

Vous pouvez commencer la classe d'une animation par une définition de ce genre :

```
public class myClass extends Sprite {
```

Toutefois, en étendant un `Sprite` plutôt qu'un `MovieClip`, vous présupposez que l'animation ne contient qu'une image. Tout code qui ferait référence aux autres images ne fonctionnerait dès lors pas comme prévu.

La fonction constructeur possède-t-elle le bon nom ?

Si vous avez une classe nommée `maClasse`, la fonction constructeur doit être très précisément nommée `maClasse`, sans quoi elle ne s'exécutera pas lorsque la classe sera initialisée. Si vous ne souhaitez pas qu'elle s'exécute immédiatement, nommez-la par exemple `lancerMaClasse` et appelez-la après que l'image démarre.

Problèmes d'exécution

Il existe aussi des problèmes qui ne provoquent pas d'erreur de compilateur et ne se font pas remarquer au départ. Ils peuvent apparaître par la suite en cours de développement et se révéler très difficiles à débusquer.

Définissez-vous des propriétés avant que l'objet ne soit prêt ?

En voilà un qui m'agace particulièrement. Vous passez à une nouvelle image dans l'animation ou un clip et vous essayez de définir une propriété d'un objet ou d'y accéder. Malheureusement, l'image et ses objets n'ayant pas encore été initialisés, la propriété concernée n'existe pas.

Les fichiers **TooEarlyExample.fla** et **TooEarlyExample.as** en livrent un bon exemple. La classe saute à l'image 2 du scénario principal, où attendent deux champs texte. Elle tente immédiatement de définir le texte du premier champ mais cette action déclenche un message d'erreur à l'exécution. Le second champ est défini lorsque l'animation a terminé l'initialisation et exécute le script dans cette image. Ce script à son tour appelle une fonction dans la classe. Cette fonction définit le texte du second champ sans problème.

Détruisez-vous des objets ?

Bien que cela ne soit pas nécessairement un gros problème, il est de bon usage de supprimer les objets que vous avez créés lorsque vous avez fini de les utiliser. Par exemple, si le joueur tire des balles dans tous les sens à l'écran, il peut maintenir une touche enfoncée et en décocher des milliers à la minute. Lorsque ces balles quittent la portion visible de l'écran, il est inutile de les conserver en mémoire et de les laisser accaparer l'attention du programme.

Pour supprimer complètement un objet, vous devez simplement vous débarrasser de toutes les références qui y renvoient dans vos variables et vos tableaux et utiliser `removeChild` pour les retirer de sa liste d'affichage.

Toutes les variables sont-elles correctement typées ?

Le typage des variables fait partie des autres facteurs qui, sans provoquer de problème sur le moment, peuvent causer bien des soucis à long terme. N'utilisez pas le type `Number` lorsque les types `int` ou même `uint` conviennent. Ce dernier type est bien plus rapide à manipuler et consomme moins de mémoire. Si vous stockez des milliers de nombres dans des tableaux, il se peut que vous constatiez un certain ralentissement si vous utilisez le type `Number` plutôt que `int`.

Il y a pire encore, si vous utilisez des variables non typées, autrement dit des variables `Object`. Ces dernières peuvent stocker des nombres et des entiers mais génèrent une surcharge significative. Veillez aussi à ne pas laisser passer les objets `MovieClip`, qu'il serait possible de remplacer par de simples objets `Sprite` à une image.

Problèmes de test

Ces problèmes sont liés à ce qui se produit durant les tests ou ce qui peut être intégré dans vos procédures de test.

Devez-vous désactiver les raccourcis clavier ?

Si vous utilisez la saisie clavier lorsque vous testez vos animations, il se peut que vous constatiez que certaines touches ne répondent pas. C'est que l'environnement de test dispose de quelques raccourcis clavier qui accaparent les touches concernées.

Pour désactiver les raccourcis clavier dans l'environnement de test et permettre à votre animation de fonctionner comme elle le ferait sur le Web, choisissez Contrôle > Désactiver les raccourcis clavier.

Avez-vous effectué des tests avec d'autres cadences d'images ?

Si vous utilisez une animation temporelle, la cadence d'images ne devrait pas importer. L'animation devrait avancer à vitesse constante. Il vaut cependant la peine d'effectuer des tests avec une cadence faible, comprise entre 6 et 12, afin de voir ce que les utilisateurs dotés d'ordinateurs lents sont susceptibles de découvrir. Nous utiliserons la technique d'animation temporelle tout au long de ce livre.

Avez-vous effectué un test sur un serveur ?

Un problème similaire se produit lorsque vous partez du principe que les objets sont tous présents au début d'une animation. En vérité, les animations Flash sont diffusées en flux continu, ce qui signifie qu'elles commencent avant que l'ensemble du contenu multimédia ait été chargé.

À la différence, lorsque vous testez une animation sur votre ordinateur local, tout le contenu multimédia est instantanément chargé. Lorsque vous chargez et testez l'animation sur un serveur, il se peut que vous constatiez qu'une portion manque pendant les quelques premières secondes, voire les premières minutes.

Lorsque vous testez une animation, vous pouvez redémarrer le test en choisissant Affichage > Simuler le téléchargement. Vous pouvez aussi choisir une option du sous-menu Affichage > Paramètres de téléchargement pour définir une vitesse de téléchargement simulé, comme 56 K. L'animation redémarre alors en simulant une mise à disposition par flux continu des objets, à la vitesse désirée. Pour ma part, je préfère toujours faire également un test avec un véritable serveur, par sécurité.

La solution à tous ces problèmes consiste à proposer un écran de téléchargement dont le rôle se limite à attendre que la totalité des éléments soit chargée. Nous examinerons un exemple d'écran de téléchargement au Chapitre 2.

Cette check-list devrait vous permettre d'éviter plus facilement les problèmes courants et donc de consacrer plus de temps à la création de vos jeux et moins à dénicher les bogues dans le code.

Maintenant que nous avons traité des notions fondamentales d'ActionScript 3.0, nous étudierons au chapitre suivant quelques exemples courts de création des blocs constructeurs que vous utiliserez pour composer vos jeux.

2

Composants de jeu ActionScript

Au sommaire de ce chapitre :

- Créer des objets visuels
- Récupérer les entrées utilisateur
- Créer une animation
- Programmer l'interaction avec l'utilisateur
- Accéder à des données externes
- Composants de jeu divers

Avant de créer des jeux complets, nous allons procéder en composant de petits morceaux. Les programmes courts de ce chapitre offrent un aperçu de certains des concepts essentiels d'ActionScript 3.0 et fournissent un certain nombre de blocs constructeurs qui pourront être utilisés par la suite et dans vos propres jeux.

Codes sources

http://flashgameu.com

A3GPU02_GameElements.zip

Créer des objets visuels

Nos quelques premiers éléments impliquent de créer et de manipuler des objets à l'écran. Nous récupérerons quelques clips de la bibliothèque, les transformerons en boutons, dessinerons des formes et du texte et apprendrons à regrouper des éléments dans des sprites.

Utiliser des clips

Lorsque vous avez un clip dans la bibliothèque et que vous souhaitez l'importer dans votre jeu, deux approches sont possibles.

La première consiste à faire glisser le clip sur la scène et à lui donner un nom d'occurrence dans l'inspecteur des propriétés. La Figure 2.1 présente un clip placé sur la scène puis nommé `myClipInstance` dans l'inspecteur des propriétés.

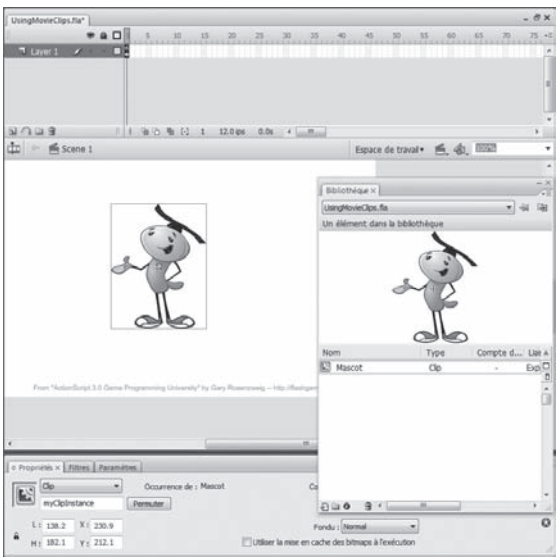


Figure 2.1
L'objet clip est nommé Mascot dans la bibliothèque, mais son occurrence dans la scène est nommée `myClipInstance`.

Ensuite, vous pouvez manipuler les propriétés du clip en faisant référence à son nom, comme ceci :

```
myClipInstance.x = 300;
myClipInstance.y = 200;
```

La seconde approche pour importer un clip dans votre jeu ne requiert que du code ActionScript. Pour commencer, vous devez cependant rendre le clip accessible en définissant ses propriétés de liaison. Sélectionnez le symbole dans la bibliothèque et, dans le menu déroulant du panneau Bibliothèque, choisissez Liaison. Cochez l'option Exporter pour ActionScript et indiquez le nom de la classe. Votre boîte de dialogue devrait ressembler à celle de la Figure 2.2.

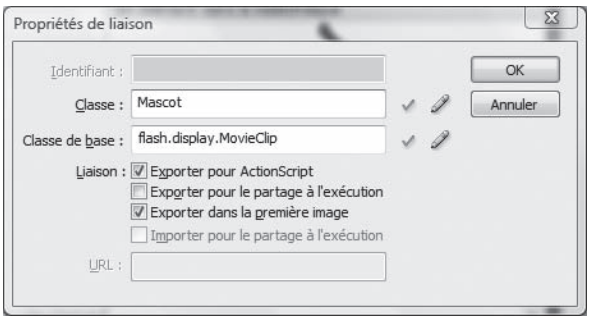


Figure 2.2
La boîte de dialogue Propriétés de liaison est paramétrée de manière que le clip Mascot puisse être utilisé par le code ActionScript.

En général, je donne à ma classe le même nom que celui du clip. Il est ainsi plus facile de le retenir.

Nous pouvons maintenant créer de nouvelles copies du clip avec ActionScript. La méthode consiste à créer une variable pour contenir l'instance[1] de l'objet et à utiliser `addChild` pour la placer dans une liste d'affichage :

```
var myMovieClip:Mascot = new Mascot();
addChild(myMovieClip);
```

Comme nous n'avons défini aucune autre propriété du clip, celui-ci apparaît à l'emplacement 0, 0 sur la scène. Nous pouvons définir son emplacement avec les propriétés `x` et `y` de l'instance. Il est aussi possible de définir son angle de rotation avec la propriété `rotation`. La valeur de cette propriété s'exprime en degrés :

```
var myMovieClip:Mascot = new Mascot();
myMovieClip.x = 275;
myMovieClip.y = 150;
myMovieClip.rotation = 10;
addChild(myMovieClip);
```

1. NdT : puisqu'il s'agit ici de code, nous reprenons l'anglicisme universellement adopté dans la terminologie des langages de programmation. Dans l'interface Flash, on parlera plus naturellement des "occurrences" d'un objet. L'anglais utilise dans les deux cas le même terme : "instance".

Si tout cela paraît bien du travail pour un simple clip, notez qu'ActionScript permet facilement d'ajouter plusieurs copies d'un clip. Le code qui suit crée ainsi dix copies de l'objet `Mascot`, en changeant l'emplacement horizontal de manière à progresser de 50 pixels vers la droite à chaque fois. Il fixe également l'échelle des clips à 50 % :

```
for(var i=0;i<10;i++) {
    var mascot:Mascot = new Mascot();
    mascot.x = 50*i+50;
    mascot.y = 300;
    mascot.scaleX = .5;
    mascot.scaleY = .5;
    addChild(mascot);
}
```

La Figure 2.3 présente le résultat de ces deux fragments de code. Le premier objet `Mascot` se trouve en haut, aux coordonnées 275, 100. Les autres objets `Mascot` sont répartis de 50 à 500 au niveau de la position verticale 300 et sont mis à l'échelle à 50 %.

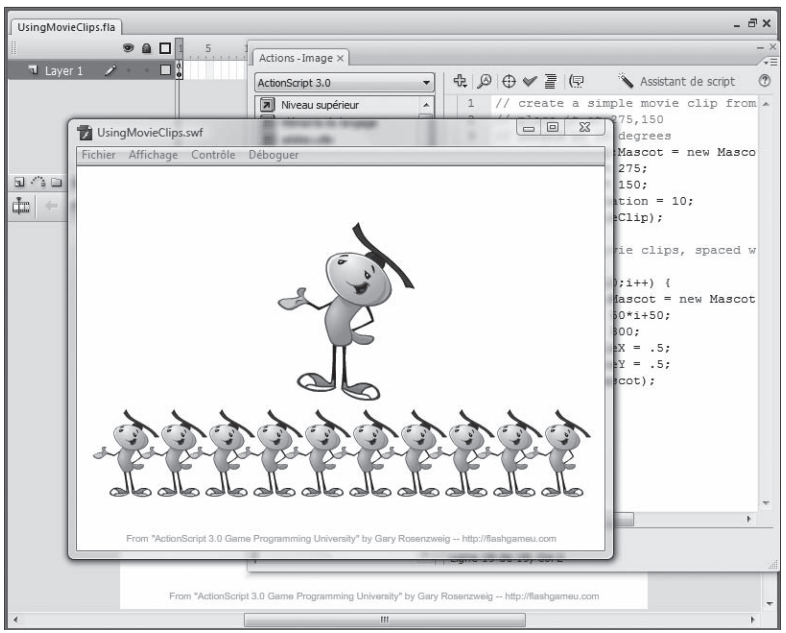


Figure 2.3
Onze mascottes ont été créées et placées par le code ActionScript.

Vous trouverez cet exemple dans l'animation **UsingMovieClips.fla**. Le code est inclus dans l'image 1.

Créer des boutons

Il est également possible de créer des boutons en n'utilisant que du code ActionScript. Ces boutons peuvent être créés à partir de clips ou de symboles de bouton stockés dans la bibliothèque.

Pour transformer un clip en un bouton sur lequel l'utilisateur peut cliquer, vous n'avez qu'à lui attribuer un écouteur. Le clip peut alors accepter des événements et notamment un événement de clic de souris.

Le code suivant place un nouveau clip aux coordonnées 100, 150 :

```
var myMovieClip:Mascot = new Mascot();
myMovieClip.x = 100;
myMovieClip.y = 150;
addChild(myMovieClip);
```

Pour attribuer un écouteur, utilisez la fonction `addEventListener`. Incluez le type d'événement auquel l'écouteur doit répondre. Il s'agit de valeurs constantes qui varient selon le type d'objet et d'événement. Dans le cas présent, `MouseEvent.CLICK` réagira à un clic avec la souris. Ensuite, incluez également une référence à la fonction que vous allez créer pour gérer l'événement (dans le cas présent, `clickMascot`) :

```
myMovieClip.addEventListener(MouseEvent.CLICK, clickMascot);
```

La fonction `clickMascot` transmet simplement un message à la fenêtre Sortie. Dans une application ou un jeu, cette fonction serait évidemment amenée à réaliser des opérations plus productives :

```
function clickMascot(event:MouseEvent) {
    trace("You clicked the mascot!");
}
```

Pour mieux donner au clip l'apparence d'un bouton, vous pouvez attribuer la valeur `true` à la propriété `buttonMode` de son instance. Le curseur se transforme alors en une main lorsque l'utilisateur le survole :

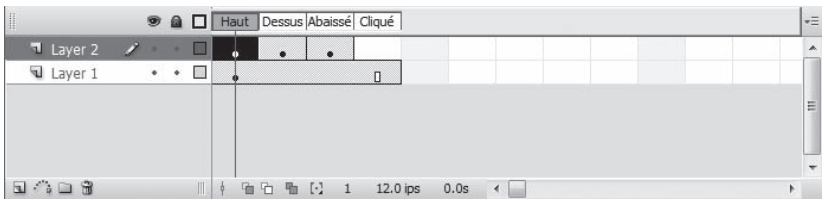
```
myMovieClip.buttonMode = true;
```

Vous pouvez évidemment aussi créer des instances de symboles de bouton avec du code ActionScript. Le principe est le même qu'avec les clips. Dans l'exemple suivant, le symbole est lié en tant que classe `LibraryButton` :

```
var myButton:LibraryButton = new LibraryButton();
myButton.x = 450;
myButton.y = 100;
addChild(myButton);
```

La principale différence entre les clips et les symboles de bouton tient à ce que les boutons possèdent quatre images spécialisées dans leur scénario. La Figure 2.4 présente le scénario de notre symbole `LibraryButton`.

Figure 2.4
Le script d'un bouton contient quatre images représentant les trois étapes du bouton et une zone réactive.

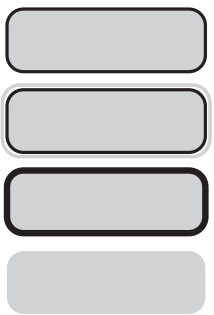


La première image représente l'apparence du bouton lorsque le curseur ne le survole pas. La seconde correspond à l'apparence du bouton lorsque le curseur le survole. La troisième image définit l'apparence du bouton lorsque l'utilisateur a cliqué dessus sans avoir encore relâché le bouton de la souris. Enfin, la dernière image correspond à la zone réactive du bouton. Elle n'est jamais visible.

La dernière image peut inclure un graphisme plus large que le reste afin de permettre à l'utilisateur de cliquer sur le bouton ou à proximité du bouton. Si les images visibles du bouton incluent des zones vides (par exemple s'il s'agit simplement de lettres) ou possèdent une forme bizarre, on peut donner à la dernière image une forme circulaire ou rectangulaire plus standard qui représente la zone réactive. Vous pouvez enfin créer des boutons invisibles en ne plaçant rien dans les images à l'exception de la dernière.

La Figure 2.5 présente les trois états de bouton et la zone réactive d'un clip. Il ne s'agit là que d'un exemple. Votre bouton peut présenter toutes sortes d'états Dessus et Abaissé.

Figure 2.5
Les quatre images qui composent un symbole de bouton.



Vous pouvez ajouter un écouteur au bouton de la même manière que vous l'avez fait avec un clip :

```
myButton.addEventListener(MouseEvent.CLICK, clickLibraryButton);
function clickLibraryButton(event:MouseEvent) {
    trace("You clicked the Library button!");
}
```

La troisième option pour créer un bouton consiste à utiliser le type `SimpleButton` afin de créer un bouton de toutes pièces. Ou presque. Il vous faut en réalité un clip pour chacune des quatre images du bouton : Haut, Dessus, Abaissé et Cliqué. Quatre éléments doivent donc figurer dans la bibliothèque au lieu d'un.

Pour créer ce type de bouton, utilisez le constructeur `SimpleButton`. Chacun des quatre paramètres de `SimpleButton` doit être une instance de clip. Dans le cas présent, nous allons utiliser quatre clips : `ButtonUp`, `ButtonOver`, `ButtonDown` et `ButtonHit` :

```
var mySimpleButton:SimpleButton = new SimpleButton(new ButtonUp(),
new ButtonOver(), new ButtonDown(), new ButtonHit());
mySimpleButton.x = 450;
mySimpleButton.y = 250;
addChild(mySimpleButton);
```

Vous pouvez également utiliser le même clip pour plusieurs des quatre paramètres de `SimpleButton`. Par exemple, vous pouvez réutiliser l'état Haut du clip pour le clip Cliqué. Vous pouvez du reste utiliser le même clip pour les quatre états. Le bouton sera moins intéressant, mais il nécessitera moins de clips dans la bibliothèque.

Vous pouvez cette fois encore ajouter un écouteur au bouton que vous venez de créer avec la commande `addEventListener` :

```
mySimpleButton.addEventListener(MouseEvent.CLICK, clickSimpleButton);
function clickSimpleButton(event:MouseEvent) {
    trace("You clicked the simple button!");
}
```

Le fichier **MakingButtons.fla** inclut le code de ces trois boutons et transmet un message différent au panneau Sortie lorsque l'utilisateur clique sur chacun d'entre eux.

Dessiner des formes

Tous les éléments à l'écran ne doivent pas nécessairement provenir de la bibliothèque. Vous pouvez utiliser ActionScript 3.0 pour tracer des lignes et des formes élémentaires.

Chaque objet d'affichage possède un calque graphique auquel vous pouvez accéder avec la propriété `graphics`. C'est le cas d'ailleurs de la scène elle-même, à laquelle vous pouvez accéder directement en écrivant du code dans le scénario principal.

Pour dessiner une ligne simple, définissez le style de ligne, accédez au point de départ de la ligne, puis tracez la ligne jusqu'au point de terminaison :

```
this.graphics.lineStyle(2,0x000000);
this.graphics.moveTo(100,200);
this.graphics.lineTo(150,250);
```

Ce code définit un style de ligne de 2 pixels de large et de couleur noire, puis trace la ligne en partant de 100, 200 pour atteindre 150, 250.

Le mot-clé `this` n'est pas indispensable mais, lorsque vous souhaitez que la ligne soit dessinée dans une instance de clip spécifique, vous devez l'indiquer en utilisant son nom. Par exemple :

```
myMovieClipInstance.graphics.lineTo(150,250);
```

Nous incluerons donc le mot-clé `this` ici afin de nous le rappeler et de rendre le code plus facile à réutiliser dans vos propres projets.

Il est également possible de créer une ligne courbe avec `curveTo`. Dans ce cas, vous devez spécifier un point de terminaison et un point d'ancrage. Ce processus peut être un peu compliqué si vous n'êtes pas familiarisé avec les méthodes de création des courbes de Bézier. Il m'a fallu quelques tentatives avant de parvenir au résultat souhaité :

```
this.graphics.curveTo(200,300,250,250);
```

Ensuite, nous allons compléter la séquence de lignes par une autre ligne droite :

```
this.graphics.lineTo(300,200);
```

Nous venons de créer le trait présenté à la Figure 2.6, composé d'une ligne droite, d'une courbe et d'une nouvelle ligne droite.

Il est aussi possible de dessiner des formes. La plus simple est le rectangle. La fonction `drawRect` prend une position pour le coin supérieur gauche puis une largeur et une hauteur :

```
this.graphics.drawRect(50,50,300,250);
```

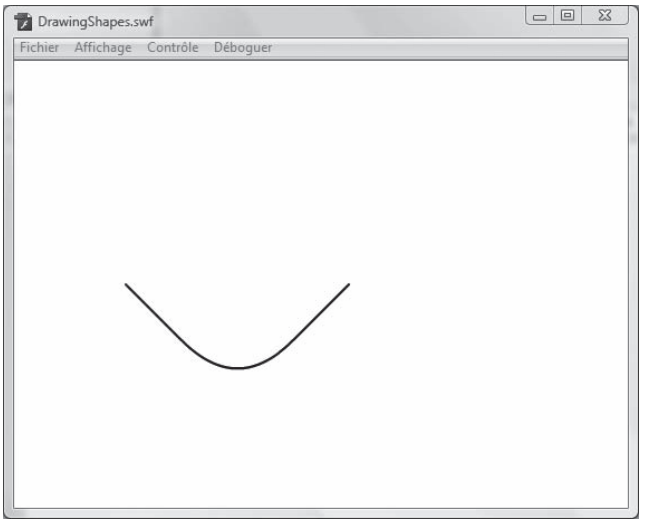


Figure 2.6
Une ligne, une courbe et une autre ligne composent ce dessin.

Vous pouvez également dessiner un rectangle à bords arrondis. Les deux paramètres supplémentaires définissent la largeur et la hauteur des bords arrondis :

```
this.graphics.drawRoundRect(40,40,320,270,25,25);
```

Il est aussi possible de créer un cercle et une ellipse. La fonction `drawCircle` prend en paramètres les coordonnées du centre et le rayon du cercle :

```
this.graphics.drawCircle(150,100,20);
```

La fonction `drawEllipse` ne procède pas de la même manière, puisqu'elle prend les mêmes paramètres de coin supérieur gauche et de taille que la fonction `drawRect` :

```
this.graphics.drawEllipse(180,150,40,70);
```

Vous pouvez encore créer des formes remplies en commençant par une fonction `beginFill` et un paramètre de couleur de remplissage :

```
this.graphics.beginFill(0x333333);
this.graphics.drawCircle(250,100,20);
```

Pour cesser d'utiliser un remplissage, exécutez la commande `endFill`.

La Figure 2.7 présente le résultat de l'ensemble des tracés que nous venons d'effectuer.

La plupart de ces fonctions de dessin possèdent d'autres paramètres. Par exemple, `lineStyle` peut aussi prendre un paramètre alpha afin de dessiner une ligne semi-transparente. Consultez la documentation pour chacune de ces fonctions si vous souhaitez en apprendre plus à ce sujet.

Vous pourrez retrouver les précédents exemples dans le fichier d'exemple **DrawingShapes.fla**.

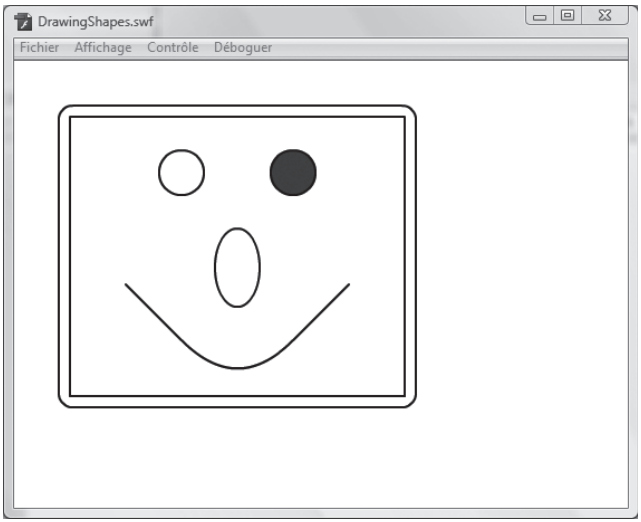


Figure 2.7
Deux lignes, une courbe, un cercle, une ellipse, un cercle rempli, un rectangle et un rectangle arrondi.

Tracer du texte

Les exemples Hello World du Chapitre 1 ont montré comment créer des objets `TextField` afin de placer du texte à l'écran. Le processus implique de créer un nouvel objet `TextField`, de définir sa propriété `text` et d'utiliser `addChild` pour l'ajouter dans la scène :

```
var myText:TextField = new TextField();
myText.text = "Check it out!";
addChild(myText);
```

Vous pouvez également définir l'emplacement du champ avec les propriétés `x` et `y` :

```
myText.x = 50;
myText.y = 50;
```

De la même manière, vous pouvez définir la largeur et la hauteur du champ :

```
myText.width = 200;
myText.height = 30;
```

Parfois, il peut être difficile d'estimer la taille d'un champ texte. Une largeur de 200 peut sembler suffisante pour contenir le texte courant, mais pourra-t-elle contenir un autre texte si vous décidez de le changer ? L'un des moyens rapides de voir la taille effective d'un champ texte consiste à positionner la propriété `border` à `true` pendant que vous effectuez vos tests :

```
myText.border = true;
```

La Figure 2.8 présente le champ texte avec une bordure afin de pouvoir en apercevoir la taille.

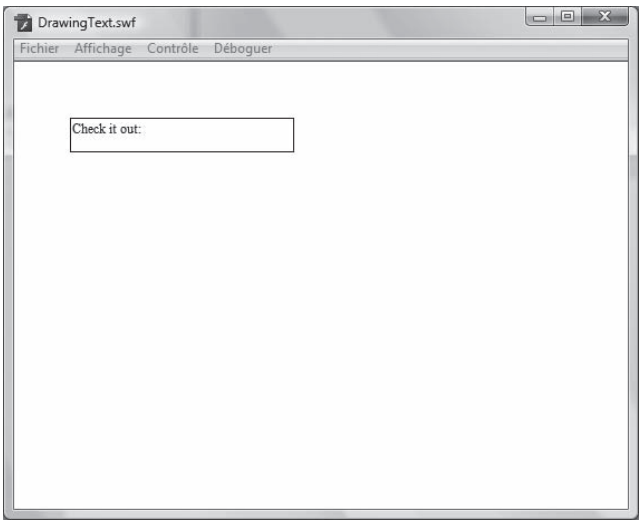


Figure 2.8
Un champ texte à 50, 50 avec une largeur de 200 et une hauteur de 30.

Il convient aussi presque toujours de s'occuper de la propriété `selectable`. Dans la plupart des cas, vous ne souhaiterez pas qu'elle soit activée, or il s'agit du réglage par défaut. Si vous ignorez cette propriété, le curseur du joueur se transforme en un curseur d'édition de texte lorsqu'il survole le texte et lui donne la possibilité de le sélectionner :

```
myText.selectable = false;
```

Lorsque vous créerez du texte, il y a toutes les chances que vous souhaiterez définir explicitement la police, la taille et le style du texte. Ces réglages ne peuvent se réaliser directement. Vous devez en fait créer un objet `TextFormat`, puis positionner ses propriétés `font`, `size` et `bold` :

```
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial";
myFormat.size = 24;
myFormat.bold = true;
```

Vous pouvez également créer un objet `TextFormat` avec une seule ligne de code. Par exemple, l'exemple précédent pourrait être réalisé de la manière suivante :

```
var myFormat:TextFormat = new TextFormat("Arial", 24, 0x000000, true);
```

La fonction constructeur `TextFormat` accepte jusqu'à treize paramètres, mais vous pouvez utiliser la valeur `null` pour ignorer tous les paramètres que vous ne souhaitez pas définir. Consultez la documentation pour en obtenir une liste complète.

Maintenant que nous avons un objet `TextFormat`, il existe deux moyens de l'utiliser. Le premier consiste à utiliser `setTextFormat` sur un objet `TextField`. Cette approche modifie le texte afin d'utiliser le style courant. Il n'est cependant pas nécessaire d'y recourir à chaque fois que vous changez la propriété `text` du champ.

L'approche préférable dans ce cas consiste à utiliser `defaultTextFormat`. Vous devez le faire avant de définir la propriété `text`. Le `text` suivant récupère alors les propriétés de style décrites dans `TextFormat`. En fait, à chaque fois que vous définissez le texte de cet objet `TextField`, vous utilisez le même style. Cette approche sera le plus souvent la meilleure pour l'usage que nous ferons des champs texte lors du développement de nos jeux :

```
myText.defaultTextFormat = myFormat;
```

La Figure 2.9 présente le champ avec son format défini.

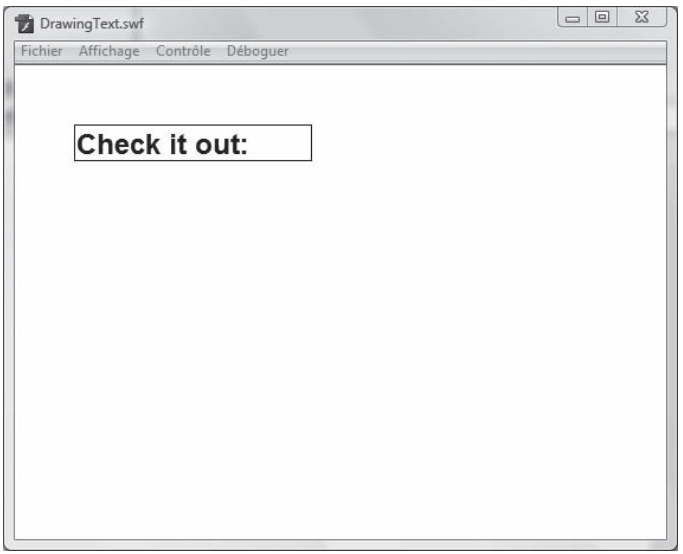


Figure 2.9
Le format texte a été défini pour une police Arial à 24 points et en gras.

Étudiez l'ensemble des propriétés de `TextFormat` dans la documentation si vous prévoyez d'étendre ses capacités. Vous pouvez également choisir d'utiliser des objets `StyleSheet` et du texte balisé en HTML à l'aide de la propriété `htmlText` de l'objet `TextField`. La fonctionnalité des feuilles de style est très étoffée : consultez la documentation si vous souhaitez en apprendre plus à ce sujet.

Créer du texte lié

Qu'obtient-on en croisant un champ texte et un bouton ? Un lien hypertexte, évidemment. Ces éléments peuvent eux aussi se créer facilement avec du code ActionScript 3.0.

Le moyen le plus simple de créer du texte lié dans un `TextField` consiste à utiliser la propriété `htmlText` du champ et à lui fournir du code HTML au lieu du texte brut utilisé par la propriété `text` :

```
var myWebLink:TextField = new TextField();
myWebLink.htmlText = "Visit <A HREF='http://flashgameu.com'>FlashGameU.com</A>!";
addChild(myWebLink);
```

Ce code fonctionne comme il le ferait dans une page Web, à ceci près qu'il n'existe pas de modification de style par défaut pour le lien. Il prend la même couleur et le même style que le reste du texte. Néanmoins, lorsque l'utilisateur clique dessus, il quitte la page Web dans son navigateur et se trouve conduit jusqu'à celle spécifiée par le lien.

Si l'animation Flash s'exécute sous la forme d'un projecteur Flash autonome, le lien lance le navigateur et conduit l'utilisateur à la page Web choisie lorsque celui-ci clique dessus. Vous pouvez également spécifier le paramètre `TARGET` de la balise `A` si vous en avez l'habitude en HTML. Utilisez `_top` pour spécifier la page entière, par opposition au cadre (`frame`), ou `_blank` pour ouvrir une nouvelle fenêtre dans le navigateur.

Si vous souhaitez que le texte apparaisse en bleu et en souligné comme il le ferait dans une page Web ordinaire, vous pouvez définir une feuille de style rapide et définir la propriété `styleSheet` avant `htmlText` :

```
var myStyleSheet:StyleSheet = new StyleSheet();
myStyleSheet.setStyle("A",{textDecoration: "underline", color: "#0000FF"});
var myWebLink:TextField = new TextField();
myWebLink.styleSheet = myStyleSheet;
myWebLink.htmlText = "Visit <A HREF='http://flashgameu.com'>FlashGameU.com</A>!";
addChild(myWebLink);
```

La Figure 2.10 présente le texte qui utilise à la fois la propriété `textFormat` avec la police Arial à 24 points et en gras et `styleSheet` pour passer le lien en bleu et le souligner.

Vos liens ne doivent pas nécessairement conduire à des pages Web. Vous pouvez les utiliser comme de simples boutons, en attribuant des écouteurs aux champs texte afin de réagir lorsque l'utilisateur clique dessus.

Pour cela, vous devez simplement utiliser `event:` dans la balise `HREF` du lien. Ensuite, fournissez du texte que votre fonction écouteur pourra recevoir :

```
myLink.htmlText = "Click <A HREF='event:testing'>here</A>";
```

Figure 2.10
Les propriétés `defaultTextFormat` et `styleSheet` ont toutes deux été utilisées pour formater le texte et le lien.

L'écouteur récupérera le texte `"testing"` sous forme de chaîne dans la propriété `text` de l'événement retourné :

```
addEventListener(TextEvent.LINK, textLinkClick);
function textLinkClick(event:TextEvent) {
    trace(event.text);
}
```

Vous pourrez ainsi définir plusieurs liens dans un `TextField` puis faire le tri pour savoir sur lequel d'entre eux l'utilisateur a cliqué en utilisant la propriété `text` du paramètre `event`. Les liens texte peuvent ainsi être utilisés à la manière de boutons.

Vous pouvez aussi formater le texte avec `defaultTextFormat` et `styleSheet` comme le lien Web. Le fichier **CreatingLinkedText.fla** inclut des exemples des deux types de liens qui utilisent le même format et le même style.

Créer des groupes de sprites

Maintenant que vous savez créer une variété d'éléments à l'écran, il est temps d'entrer un peu plus en matière et de voir comment fonctionnent les objets d'affichage et les listes d'affichage. Il est possible de créer des objets d'affichage `Sprite`, qui n'ont d'autre rôle que de contenir d'autres objets d'affichage.

Le code qui suit crée un nouveau `Sprite` et dessine un rectangle de 200 par 200 à l'intérieur. Le rectangle se voit attribuer une bordure noire de 2 pixels de large et un remplissage gris clair :

```
var sprite1:Sprite = new Sprite();
sprite1.graphics.lineStyle(2,0x000000);
sprite1.graphics.beginFill(0xCCCCCC);
sprite1.graphics.drawRect(0,0,200,200);
addChild(sprite1);
```

Le `Sprite` peut ensuite être positionné, avec la forme que nous avons dessinée à l'intérieur, aux coordonnées 50, 50 dans la scène :

```
sprite1.x = 50;
sprite1.y = 50;
```

Nous allons maintenant créer un second `Sprite`, comme le premier, mais en le positionnant cette fois à 300, 100 :

```
var sprite2:Sprite = new Sprite();
sprite2.graphics.lineStyle(2,0x000000);
sprite2.graphics.beginFill(0xCCCCCC);
sprite2.graphics.drawRect(0,0,200,200);
sprite2.x = 300;
sprite2.y = 50;
addChild(sprite2);
```

Créons pour finir un troisième `Sprite`, qui contiendra cette fois un cercle. Au lieu d'utiliser `addChild` pour le placer dans la scène, nous le placerons à l'intérieur de `sprite1`. Nous lui attribuerons en outre un remplissage plus foncé :

```
var sprite3:Sprite = new Sprite();
sprite3.graphics.lineStyle(2,0x000000);
sprite3.graphics.beginFill(0x333333);
sprite3.graphics.drawCircle(0,0,25);
sprite3.x = 100;
sprite3.y = 100;
sprite1.addChild(sprite3);
```

La Figure 2.11 montre à quoi ressemblent nos trois sprites à l'écran. Vous remarquerez que, bien que nous ayons fixé les propriétés `x` et `y` du cercle aux coordonnées 100, 100, ce cercle apparaît à cet emplacement non pas relativement à la scène mais relativement au contenu de `sprite1`.

L'animation contient maintenant `sprite1` et `sprite2` comme enfants de la scène. `sprite3` est pour sa part un enfant de `sprite1`. Si nous amenons `sprite3` à devenir un enfant de `sprite2`, il saute au centre de `sprite2` car la position 100, 100 est dès lors relative à `sprite3`, son nouveau parent.

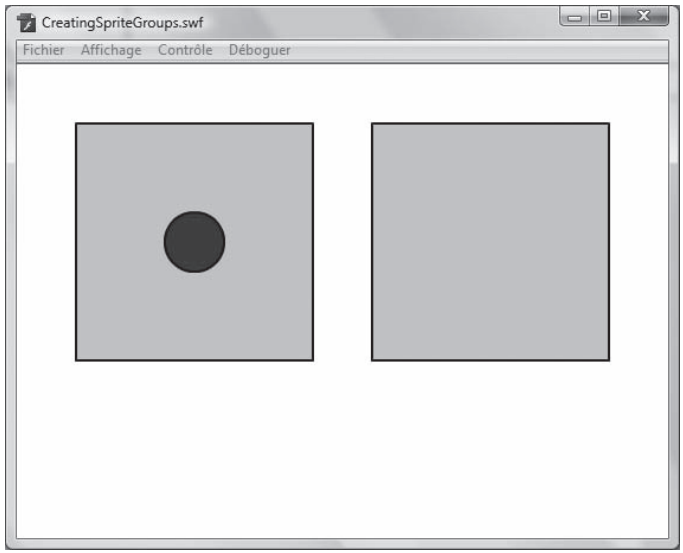


Figure 2.11
Le sprite du cercle est à l'intérieur du sprite du rectangle de gauche.

L'animation **CreatingSpriteGroups.fla** permet de visualiser ce mécanisme plus facilement en plaçant un écouteur sur `sprite1` et sur `sprite2`. Lorsque vous cliquez sur l'un d'entre eux, `sprite3` devient son enfant. Vous pouvez ainsi faire sauter `sprite3` d'un parent à l'autre :

```
sprite1.addEventListener(MouseEvent.CLICK, clickSprite);
sprite2.addEventListener(MouseEvent.CLICK, clickSprite);
function clickSprite(event:MouseEvent) {
    event.currentTarget.addChild(sprite3);
}
```

Cet exemple montre bien également comment un écouteur de bouton peut être utilisé pour plusieurs boutons. L'objet sur lequel l'utilisateur clique est passé à la fonction écouteur via `currentTarget`. Dans le cas présent, nous utilisons cette valeur pour `addChild`. Vous pouvez cependant aussi la comparer à une liste d'objets sur lesquels l'utilisateur est susceptible de cliquer et exécuter du code en fonction de l'objet désigné.

Lors du développement de nos jeux, nous ne cesserons de créer des groupes de `Sprite` pour contenir différents types d'éléments de jeu. Si nous utilisons des `Sprite` simplement pour les calques, nous les conserverons tous à 0, 0 et pourrons déplacer des éléments de `Sprite` en `Sprite` sans changer leur position relative à l'écran.

Définir la profondeur du Sprite

Le moment est sans doute bien choisi pour signaler maintenant la commande `setChildIndex`, qui permet de déplacer les objets d'affichage vers le haut et le bas dans la liste d'affichage. Elle offre en d'autres termes la possibilité de placer un `Sprite` au-dessus d'un autre.

La liste d'affichage peut se comparer à un tableau qui commence par l'élément 0. Si vous avez créé trois `Sprite`, ceux-ci se trouvent aux positions 0, 1 et 2. La position 2 correspond au `Sprite` du haut, qui est dessiné au-dessus des autres.

Si vous souhaitez déplacer un `Sprite` vers le bas, autrement dit sous les autres `Sprite`, utilisez simplement la commande suivante :

```
setChildIndex(myMovieClip,0);
```

Ce code place l'objet d'affichage `myMovieClip` à la position 0, tandis que tous les autres remontent d'un cran pour remplir l'espace laissé libre par l'objet déplacé.

Il est un petit peu plus compliqué de placer un `Sprite` au-dessus des autres. Vous devez attribuer à l'index la valeur du dernier élément dans la liste d'affichage. S'il y a trois éléments (0, 1 et 2), vous devez donc lui attribuer la valeur 2. Vous pouvez le faire à l'aide de la propriété `numChildren` :

```
setChildIndex(myMovieClip,numChildren-1);
```

Vous devez utiliser `-1` parce que, s'il existe trois enfants (0, 1 et 2), `numChildren` retourne 3. Or il faut ici utiliser 2 dans `setChildIndex`. Le nombre 3 générerait une erreur.

L'animation d'exemple **SettingSpriteDepth.fla** place à l'écran trois `Sprite` qui se superposent les uns aux autres. Vous pouvez cliquer sur chacun d'entre eux pour les amener au premier plan.

Accepter les entrées de l'utilisateur

Les sections qui suivent traitent de la récupération des entrées du joueur. Elles proviennent toujours du clavier ou de la souris car il s'agit des seuls périphériques de saisie standard sur les ordinateurs modernes.

Entrée souris

Nous savons déjà bien comment transformer un `Sprite` en un bouton et l'amener à réagir aux clics de souris. La souris ne sert cependant pas qu'à cliquer. Vous pouvez aussi récupérer la position du curseur à tout moment et les `Sprite` peuvent détecter si le curseur les survole.

Pour déterminer à tout moment l'emplacement sur la scène du curseur, utilisez les propriétés `mouseX` et `mouseY`. Le code qui suit récupère l'emplacement courant du curseur et le place dans un champ texte à chaque image :

```
addEventListener(Event.ENTER_FRAME, showMouseLoc);
function showMouseLoc(event:Event) {
    mouseLocText.text = "X="+mouseX+" Y="+mouseY;
}
```

Pour détecter le moment où le curseur survole un `Sprite`, vous pouvez procéder de la même manière que lorsque vous détectez un clic de souris. Au lieu d'un clic, vous devez cependant rechercher un événement `rollover`. Un écouteur destiné à cela peut être ajouté au `Sprite` :

```
mySprite.addEventListener(MouseEvent.ROLL_OVER, rolloverSprite);
function rolloverSprite(event:MouseEvent) {
    mySprite.alpha = 1;
}
```

Dans cette fonction, nous attribuons la valeur 1 à la propriété `alpha` du `Sprite`, ce qui le rend opaque à 100 %. Ensuite, lorsque le curseur quitte le `Sprite`, nous la réduisons à 50 % :

```
mySprite.addEventListener(MouseEvent.ROLL_OUT, rolloutSprite);
function rolloutSprite(event:MouseEvent) {
    mySprite.alpha = .5;
}
```

Dans l'animation **MouseInput.fla**, le `Sprite` commence à 50 % d'opacité et ne passe à 100 % que lorsque le curseur le survole. La Figure 2.12 présente les valeurs dans le champ texte de l'emplacement du curseur et ce `Sprite`.

Entrée clavier

La détection des entrées clavier s'appuie sur les deux événements clavier `KEY_UP` et `KEY_DOWN`. Lorsque l'utilisateur enfonce une touche, le message `KEY_DOWN` est envoyé. Si vous définissez un écouteur pour le surveiller, vous pouvez en tirer parti.

La fonction `addEventListener` doit cependant référencer l'objet scène. En effet, les appuis sur les touches n'ont pas de cible évidente comme les clics de souris. Un objet doit donc recevoir le focus clavier au démarrage de l'animation. C'est à la scène que revient ce privilège :

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownFunction);
```

Lorsqu'une fonction récupère l'appel de cet écouteur, elle peut accéder à plusieurs propriétés du paramètre de l'événement. L'un de ces paramètres est `charCode`, qui retourne le numéro de caractère de la touche enfoncée.

Dans l'exemple suivant, le `charCode` est converti en un caractère puis affiché dans le champ texte `keyboardText` :

```
function keyDownFunction(event:KeyboardEvent) {
    keyboardText.text = "Key Pressed: "+String.fromCharCode(event.charCode);
}
```

N'oubliez pas de sélectionner Contrôle > Désactiver les raccourcis clavier au moment de vos tests. Sans cela, vos appuis sur les touches pourraient ne pas parvenir jusqu'à la scène.

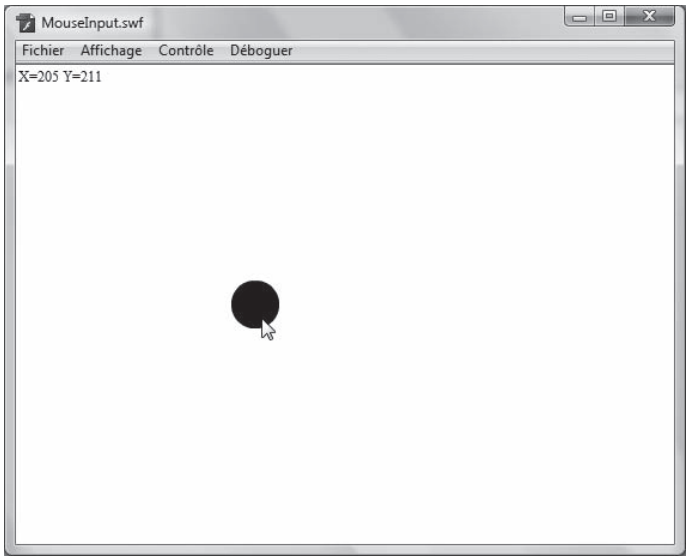


Figure 2.12
Le curseur survole le sprite, qui devient donc opaque.

Parmi les propriétés de l'événement figure également `keyCode`, qui s'apparente à `charCode` mais n'est pas affectée par la touche Maj. Par exemple, lorsque la touche Maj est enfoncée, la touche A donne le `charCode` 65 pour un A majuscule. Lorsque la touche est relâchée, le `charCode` est 97, ce qui représente un a minuscule. À la différence, `keyCode` retourne 65 dans les deux cas.

Parmi les autres propriétés, on peut encore citer `ctrlKey`, `shiftKey` et `altKey`, qui indiquent si ces touches de modification sont enfoncées.

Dans les jeux, on ne se soucie généralement pas de l'appui initial sur la touche, mais du fait que le joueur continue à la maintenir enfoncée. Par exemple, dans un jeu de conduite, il faut savoir si le joueur conserve l'accélérateur enfoncé, que représente la touche fléchée du haut.

Pour reconnaître qu'une touche est enfoncée, la stratégie consiste à rechercher à la fois `KEY_DOWN` et `KEY_UP`. Si nous détectons qu'une touche est enfoncée, nous positionnons une variable booléenne correspondant à `true`. Ensuite, lorsque la même touche est relâchée, nous la positionnons à `false`. Pour déterminer à tout moment si la touche est enfoncée, il suffit dès lors de vérifier la valeur de la variable booléenne.

Voici du code qui teste la barre d'espace de cette manière. La première fonction repère à quel moment la barre d'espace est enfoncée et positionne la variable `spacePressed` à `true` :

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownFunction);
function keyDownFunction(event:KeyboardEvent) {
    if (event.charCode == 32) {
        spacebarPressed = true;
    }
}
```

La fonction suivante capture le relâchement d'une touche. S'il s'agit de la barre d'espace, `keyPressed` est positionnée à `false` :

```
stage.addEventListener(KeyboardEvent.KEY_UP, keyUpFunction);
function keyUpFunction(event:KeyboardEvent) {
    if (event.charCode == 32) {
        spacebarPressed = false;
    }
}
```

Cette méthode permet de surveiller les touches essentielles pour le jeu, comme la barre d'espace et les quatre touches fléchées. L'animation d'exemple **KeyboardInput.fla** surveille la barre d'espace de cette manière et affiche un message lorsque son état change.

Entrée texte

Parmi les autres types d'objets `TextField` figure le champ de saisie. La différence entre un champ texte statique ou dynamique et le champ de saisie tient à ce que l'utilisateur peut sélectionner et taper du texte dans le champ de saisie. Pour créer un `TextField` qui agit comme un champ de saisie, définissez simplement sa propriété `type` :

```
var myInput:TextField = new TextField();
myInput.type = TextFieldType.INPUT;
addChild(myInput);
```

Ce code crée un champ de saisie difficile à repérer et bien mal formé dans le coin supérieur gauche de l'écran. Il est cependant possible de l'améliorer en définissant ses propriétés et en utilisant un objet `TextFormat`.

Le code qui suit fixe le format à 12 points dans la police Arial, positionne le champ à 10, 10 avec une hauteur de 18 et une largeur de 200. Il active également la bordure comme on s'attendrait à en voir pour n'importe quel champ de saisie dans un logiciel classique :

```
var inputFormat:TextFormat = new TextFormat();
inputFormat.font = "Arial";
inputFormat.size = 12;

var myInput:TextField = new TextField();
myInput.type = TextFieldType.INPUT;
myInput.defaultTextFormat = inputFormat;
myInput.x = 10;
myInput.y = 10;
myInput.height = 18;
```

```
myInput.width = 200;
myInput.border = true;
addChild(myInput);
stage.focus = myInput;
```

La dernière ligne de code place le curseur de saisie de texte dans le champ.

En général, les `TextField` sont configurés pour ne représenter qu'une seule ligne de texte. Cette particularité peut cependant être modifiée avec la propriété `multiline`. Pour la plupart des entrées texte, vous n'aurez toutefois besoin que d'une seule ligne. Cela signifie que les touches Retour/Entrée ne seront pas reconnues, car il n'est pas possible de créer une seconde ligne de texte. Cette touche peut néanmoins être capturée et utilisée pour signaler la fin de la saisie.

Pour capturer la touche Retour, vous devez placer un écouteur sur l'événement de relâchement de touche. Ensuite, la fonction répondante doit vérifier si la touche enfoncée possède le numéro de code 13, qui correspond à la touche Retour :

```
myInput.addEventListener(KeyboardEvent.KEY_UP, checkForReturn);
function checkForReturn(event:KeyboardEvent) {
    if (event.charCode == 13) {
        acceptInput();
    }
}
```

La fonction `acceptInput` prend le texte du champ de saisie et le stocke dans `theInputText`. Ensuite, elle le transmet à la fenêtre Sortie et supprime le champ texte :

```
function acceptInput() {
    var theInputText:String = myInput.text;
    trace(theInputText);
    removeChild(myInput);
}
```

En testant cette animation, je me suis rendu compte que l'environnement de test interceptait parfois la touche Retour, même en cochant l'option Désactiver les raccourcis clavier dans la barre des menus. Dans ce cas, cliquez sur la fenêtre et essayez à nouveau pour obtenir le résultat désiré. Ce problème ne doit pas se poser lorsque l'animation est déployée sur le Web.

L'animation d'exemple **TextInput.fla** contient le code précédent que vous pouvez tester directement.

Créer une animation

Nous allons maintenant examiner du code ActionScript qui permet de déplacer des `Sprite` à l'écran. Nous étudierons à cette occasion quelques méthodes qui permettent à ce mouvement d'imiter des déplacements réels.

Mouvement des sprites

Pour changer la position d'un `Sprite` ou d'un clip, rien de plus simple : il suffit de définir sa position `x` ou `y`. Pour animer l'un d'entre eux, il suffit donc de les modifier à intervalles réguliers.

Grâce à l'événement `ENTER_FRAME`, il est possible de programmer facilement ce type de changement régulier. Par exemple, voici un programme court qui crée une copie d'un clip dans la bibliothèque et le déplace d'un pixel vers la droite à chaque image :

```
var hero:Hero = new Hero();
hero.x = 50;
hero.y = 100;
addChild(hero);

addEventListener(Event.ENTER_FRAME, animateHero);
function animateHero(event:Event) {
    hero.x++;
}
```

Le personnage du héros se déplace maintenant dans la scène, en avançant d'un pixel à la fois. Au lieu de progresser d'un pixel à chaque fois, vous pourriez cependant avancer de 10 avec `+= 10` au lieu de `++`.

L'autre moyen de faire accélérer le héros consiste à augmenter simplement la cadence d'images. Au lieu des 12 ips (images par seconde) par défaut, vous pourriez par exemple passer à 60 ips. Cette modification s'opère dans le coin supérieur gauche de l'inspecteur des propriétés lorsque la scène est sélectionnée. La Figure 2.13 présente l'inspecteur des propriétés lorsque la cadence d'images est fixée à 12 ips.

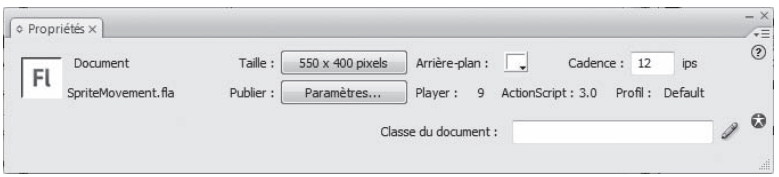


Figure 2.13
L'inspecteur des propriétés permet de modifier la cadence d'images de l'animation.

Que vous choisissiez une cadence d'images de 60 ips n'implique pas nécessairement que l'animation tournera à 60 ips. Elle tentera seulement de le faire. S'il se passe beaucoup de choses dans cette animation et si l'ordinateur de l'utilisateur est plutôt lent, il n'est pas sûr que la cadence réelle atteigne les 60 ips. Nous traiterons sous peu des animations temporelles, qui constituent une bonne alternative aux animations à images.

Au lieu de nous contenter de faire glisser le héros à l'écran, nous pouvons le faire marcher. Il nous faut cependant l'aide d'un artiste animateur. Celui-ci doit créer plusieurs images correspondant à un cycle de marche et les placer dans des images séquentielles du clip `Hero`. La Figure 2.14 présente ce cycle de marche.

Figure 2.14
Cycle de marche simple décomposé en sept images.

Il ne nous reste plus qu'à réécrire la fonction `animateHero` afin que le personnage se déplace de l'image 2 à l'image 8 du clip, qui représentent les sept images de l'animation. L'image 1 est réservée à la position statique debout.

L'artiste animateur nous indique également que le personnage doit avancer de 7 pixels par image sur le plan horizontal afin que l'animation de la marche soit naturelle.

Le code résultant vérifie la propriété `currentFrame` du clip et, s'il se trouve à l'image 8, le ramène à l'image 2. Sans cela, le clip passerait simplement à l'image suivante :

```
function animateHero(event:Event) {
    hero.x += 7;
    if (hero.currentFrame == 8) {
    hero.gotoAndStop(2);
    } else {
    hero.gotoAndStop(hero.currentFrame+1);
    }
}
```

Testez l'animation d'exemple **SpriteMovement.fla** pour voir ce code en action. Testez en outre différentes cadences d'images pour le voir avancer plus ou moins rapidement.

Utiliser des Timer

Les `Timer` (minuteurs) peuvent se comparer à de petites horloges à messages. Vous en créez une et la lancez, puis elle se met à cliquer et transmet des messages à intervalles définis. Par exemple, vous pouvez créer un `Timer` pour appeler une fonction spécifique toutes les secondes.

Pour définir un `Timer`, vous devez créer un nouvel objet `Timer`. Vous devez lui passer le nombre de millisecondes entre les événements. Vous pouvez également passer un second paramètre pour le nombre d'événements à générer avant l'arrêt, mais nous ne nous en servirons pas ici.

Le code qui suit crée un nouveau `Timer` qui déclenche un événement toutes les 1 000 millisecondes (à chaque seconde). Il appelle la fonction `timerFunction` pour chacun de ces événements :

```
var myTimer:Timer = new Timer(1000);
myTimer.addEventListener(TimerEvent.TIMER, timerFunction);
```

Pour tester le `Timer`, il suffit de l'amener à dessiner un petit cercle à chaque événement. Le paramètre `event` transmis dans la fonction inclut une propriété `target` qui fait référence au `Timer`. Vous pouvez l'utiliser pour accéder à la propriété `currentCount`, qui contient le nombre de fois où le `Timer` a été déclenché. Nous utiliserons cette technique pour décaler chaque cercle et dessiner une ligne de cercles de gauche à droite :

```
function timerFunction(event:TimerEvent) {
    this.graphics.beginFill(0x000000);
    this.graphics.drawCircle(event.target.currentCount*10,100,4);
}
```

Le fait de créer le `Timer` et d'attacher un écouteur ne suffit pas. Vous devez également demander au `Timer` de démarrer. Pour cela, utilisez la commande `start()` :

```
myTimer.start();
```

L'animation **UsingTimers.fla** illustre le fonctionnement du code précédent.

Vous pouvez également utiliser un `Timer` pour réaliser les mêmes tâches que celles présentées dans la précédente section avec les événements `enterFrame`. Voici un `Timer` qui appelle la même fonction `animateHero` pour déplacer le personnage dans l'écran en lui faisant suivre un cycle de marche. Il remplace l'appel `addEventListener` :

```
var heroTimer:Timer = new Timer(80);
heroTimer.addEventListener(TimerEvent.TIMER, animateHero);
heroTimer.start();
```

Ce code est illustré dans le fichier **UsingTimers2.fla**. Lorsque vous l'exécutez, le personnage marche à une allure correspondant à une cadence de 12 ips. Vous pouvez cependant fixer la cadence d'images à 12, 6 ou 60 : la marche s'effectuera toujours à la même vitesse.

Essayez de fixer la cadence d'images à 1 ips. Avec le `Timer` qui déplace le personnage toutes les 80 millisecondes, ce dernier fera bien du chemin entre les mises à jour de l'écran. Cet exemple montre que les `Timer` peuvent être utilisés pour créer un mouvement identique sur tous les ordinateurs jusqu'aux plus lents pour autant que les calculs réalisés avec chaque événement `Timer` ne surmènent pas le processeur.

Animation temporelle

Les animations en temps réel impliquent que les étapes de l'animation tiennent compte du temps écoulé et non d'intervalles temporels arbitraires.

Une étape d'animation temporelle doit d'abord calculer le temps écoulé depuis la dernière étape. Ensuite, elle déplace les objets en fonction de cette durée calculée. Par exemple, si le premier intervalle de temps est 0,1 seconde et le second, 0,2, les objets vont deux fois plus loin après le second intervalle de temps de manière à opérer une progression continue.

La première chose à faire est de créer une variable qui contient le temps de la dernière étape. Nous commencerons par placer la mesure temporelle courante récupérée à partir de la fonction système `getTimer()`. Cette fonction retourne le temps en millisecondes depuis que le lecteur Flash a démarré :

```
var lastTime:int = getTimer();
```

Ensuite, nous allons créer un écouteur d'événements lié à l'événement `ENTER_FRAME` qui appelle `animateBall` :

```
addEventListener(Event.ENTER_FRAME, animateBall);
```

La fonction `animateBall` calcule la différence temporelle et positionne la variable `lastTime` afin de préparer l'étape suivante. Elle définit ensuite l'emplacement `x` d'une instance de clip appelée `ball`. Elle ajoute `timeDiff` multiplié par 0,1. Le clip se déplace ainsi de 100 pixels toutes les 1 000 millisecondes :

```
function animateBall(event:Event) {
    var timeDiff:int = getTimer()-lastTime;
    lastTime += timeDiff;
    ball.x += timeDiff*.1;
}
```

L'animation **TimeBasedAnimation.fla** utilise ce code pour déplacer une balle à l'écran. Commencez par la tester avec une cadence d'images de 12 ips. Ensuite, testez-la à 60 ips. Vous remarquerez que la balle parvient de l'autre côté de l'écran au même moment, mais que le mouvement paraît bien plus fluide à 60 ips.

Animation physique

Avec les animations ActionScript, vous pouvez faire bien plus qu'amener un objet à se déplacer le long d'un chemin prédéfini. Vous pouvez aussi lui donner des propriétés physiques et le faire bouger comme un objet réel.

L'animation physique peut être à images ou temporelle. Nous allons poursuivre avec un exemple d'animation temporelle, mais en utilisant la vélocité et la gravité pour indiquer l'endroit vers lequel l'objet doit se déplacer.

La gravité est une accélération constante vers le sol (dans le cas présent, vers le bas de l'écran). Dans la réalité, la gravité est de 9,8 mètres/seconde ou de 32 pieds/seconde. Dans l'univers du lecteur Flash, tout se mesure en pixel par milliseconde. Une mise à l'échelle doit donc être effectuée par rapport au monde réel. Par exemple, si 1 pixel correspond à 1 mètre, 0,0098 correspond à 0,0098 mètre/milliseconde ou 9,8 mètres/seconde. Vous pouvez cependant tout aussi bien utiliser 0,001 ou 7 ou bien encore tout autre nombre, tant que ce réglage paraît naturel dans votre jeu. L'idée est de créer non pas des simulations scientifiques mais des jeux.

Nous fixerons la gravité à 0,0098 et définirons une vélocité de départ pour l'élément mouvant. La vélocité désigne tout simplement la vitesse et la direction d'un objet en mouvement. `dx` et `dy`, qui représentent le changement des positions horizontale et verticale définissent ensemble la vélocité :

```
// Définition de la gravité
var gravity:Number = .00098;
var b:Number = .05;
// Définition de la vélocité de départ
var dx:Number = .2;
var dy:Number = -.8;
```

L'objet (dans le cas présent, une balle) doit donc se déplacer de 0,2 pixel à l'horizontale toutes les millisecondes et de –0,8 pixel verticalement chaque milliseconde. Autrement dit, il est lancé vers le haut et la droite.

Pour contrôler l'animation, nous allons créer un écouteur `ENTER_FRAME` et initialiser la variable `lastTime` :

```
// Marquer l'instant de départ et ajouter un écouteur
var lastTime:int = getTimer();
addEventListener(Event.ENTER_FRAME, animateBall);
```

La fonction `animateBall` commence par calculer le temps écoulé depuis la dernière étape de l'animation :

```
// Animation par étapes
function animateBall(event:Event) {

    // Calculer le temps écoulé
    var timeDiff:int = getTimer()-lastTime;
    lastTime += timeDiff;
```

La variable `dy` définit la vitesse verticale et doit changer selon la traction de la gravité mesurée par la différence temporelle :

```
    // Ajuster la vitesse verticale pour la gravité
    dy += gravity*timeDiff;
```

La balle se déplace en fonction de deux variables : `dx` et `dy`. Dans les deux cas, la différence temporelle (`timeDiff`) est utilisée pour déterminer la distance :

```
    // Déplacer la balle
    ball.x += timeDiff*dx;
    ball.y += timeDiff*dy;
}
```

Si vous exécutez l'animation **PhysicsBasedAnimation.fla**, vous obtiendrez un résultat comparable à celui de la Figure 2.15.

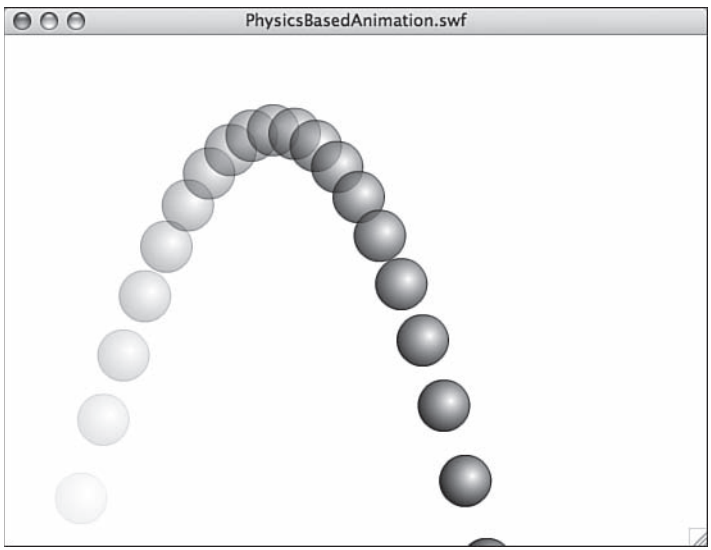


Figure 2.15
Cette capture d'écran lente fait apparaître les différentes positions de la balle à 12 ips.

Programmer l'interaction avec l'utilisateur

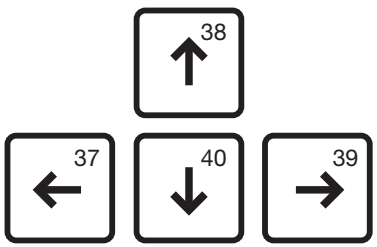
Au-delà de l'entrée utilisateur et du mouvement des sprites, il est encore possible de combiner les deux : lorsque l'interaction de l'utilisateur affecte les éléments à l'écran. Les programmes suivants sont de petits exemples d'interaction de l'utilisateur avec des sprites.

Déplacer des sprites

Les sprites à l'écran se déplacent généralement avec la souris ou le clavier. Pour le clavier, ce sont le plus souvent les touches fléchées qui servent à contrôler le sprite.

Précédemment dans ce chapitre, vous avez vu comment déterminer si la barre d'espace a été enfoncée. Il est possible d'utiliser la même procédure pour déterminer si les touches fléchées sont enfoncées. Bien que ces dernières ne possèdent pas de représentation visible sous forme de caractère, elles peuvent être représentées par les codes de touche 37, 38, 39 et 40. La Figure 2.16 présente les quatre touches fléchées et leurs codes correspondants.

Figure 2.16
Les quatre touches fléchées peuvent être référencées par ces quatre codes de touche.



Nous commençons par créer quatre variables booléennes pour y stocker l'état des quatre touches fléchées :

```
// Initialiser les variables des touches fléchées
var leftArrow:Boolean = false;
var rightArrow:Boolean = false;
var upArrow:Boolean = false;
var downArrow:Boolean = false;
```

Il nous faut à la fois des écouteurs `KEY DOWN` et `KEY UP`, ainsi qu'un écouteur `ENTER FRAME` pour gérer le déplacement du sprite à chaque mise à jour de l'écran :

```
// Définition des écouteurs événementiels
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyPressedDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyPressedUp);
stage.addEventListener(Event.ENTER_FRAME, moveMascot);
```

Lorsque l'utilisateur enfonce une touche fléchée, nous positionnons sa variable booléenne à `true` :

```
// Positionnement à true des variables des touches fléchées
function keyPressedDown(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = true;
    } else if (event.keyCode == 39) {
        rightArrow = true;
    } else if (event.keyCode == 38) {
        upArrow = true;
    } else if (event.keyCode == 40) {
        downArrow = true;
    }
}
```

De la même manière, lorsque l'utilisateur relâche les touches fléchées, nous positionnons la variable booléenne correspondante à `false` :

```
// Positionnement à false des variables des touches fléchées
function keyPressedUp(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = false;
    } else if (event.keyCode == 39) {
        rightArrow = false;
    } else if (event.keyCode == 38) {
        upArrow = false;
    } else if (event.keyCode == 40) {
        downArrow = false;
    }
}
```

Nous pouvons maintenant utiliser ces variables booléennes pour déplacer le clip de la mascotte d'une quantité définie dans la direction appropriée. Nous stockerons la quantité de mouvement dans la variable `speed`, au lieu de la répéter quatre fois dans le code :

```
// Déplacement à chaque image
function moveMascot(event:Event) {
    var speed:Number = 5;

    if (leftArrow) {
        mascot.x -= speed;
    }
    if (rightArrow) {
        mascot.x += speed;
    }
    if (upArrow) {
        mascot.y -= speed;
```

```
        }
        if (downArrow) {
            mascot.y += speed;
        }
    }
```

L'animation **MovingSprites.fla** présente ce code en action. Vous remarquerez que, puisque nous testons séparément chacune des variables booléennes des touches fléchées, il est possible de combiner ces touches. Par exemple, vous pouvez appuyer sur les touches de droite et du bas pour faire avancer la mascotte vers le bas et la droite à la fois. Si vous maintenez les touches gauche et droite simultanément enfoncées, la mascotte ne se déplace pas (les deux mouvements s'annulant).

Faire glisser des sprites

L'un des autres moyens de déplacer un `Sprite` dans la scène consiste à permettre à l'utilisateur de cliquer dessus et de le faire glisser.

Au lieu de surveiller le clavier, nous surveillerons donc cette fois la souris. Lorsque l'utilisateur clique sur le sprite, nous faisons commencer le glissement. Lorsque l'utilisateur relâche le bouton de la souris, nous l'interrompons.

Nous ne pouvons cependant pas compter sur le fait que le curseur se trouve sur le sprite lorsque l'utilisateur relâche le bouton. Nous allons donc tester l'événement `MOUSE_DOWN` sur le sprite `mascot`, mais l'événement `MOUSE_UP` sur la scène.

La scène récupère en effet un événement `MOUSE_UP` que le curseur se trouve ou non sur le sprite :

```
// Définition des écouteurs
mascot.addEventListener(MouseEvent.MOUSE_DOWN, startMascotDrag);
stage.addEventListener(MouseEvent.MOUSE_UP, stopMascotDrag);
mascot.addEventListener(Event.ENTER_FRAME, dragMascot);
```

L'autre facteur à prendre en compte est le décalage du curseur. Nous souhaitons permettre à l'utilisateur de faire glisser le sprite en saisissant n'importe quel point du sprite. Si le joueur clique sur le coin inférieur droit du sprite, le curseur et le coin inférieur droit continuent de conserver la même position relative pendant le glissement.

Pour cela, nous allons déterminer le décalage entre l'emplacement 0, 0 du sprite et l'emplacement du clic de souris et le stocker dans `clickOffset`. Nous utiliserons aussi cette variable pour déterminer si un glissement se produit à ce moment. Si c'est le cas, `clickOffset` se verra attribuer un objet `Point`. Sinon il sera `null` :

```
// Décalage entre l'emplacement du sprite et le clic
var clickOffset:Point = null;
```

Lorsque l'utilisateur clique sur le sprite, le décalage est récupéré à partir des propriétés `localX` et `localY` de l'événement de clic :

```
// L'utilisateur a cliqué
function startMascotDrag(event:MouseEvent) {
    clickOffset = new Point(event.localX, event.localY);
}
```

Lorsque l'utilisateur relâche le curseur, le `clickOffset` est ramené à `null` :

```
// L'utilisateur a relâché le bouton de la souris
function stopMascotDrag(event:MouseEvent) {
    clickOffset = null;
}
```

Ensuite, à chaque image, si `clickOffset` n'est pas `null`, nous allons définir la position de la mascotte en lui attribuant l'emplacement courant du curseur auquel nous soustrayons le décalage :

```
// Exécuter à chaque image
function dragMascot(event:Event) {
    if (clickOffset != null) { // must be dragging
        mascot.x = mouseX - clickOffset.x;
        mascot.y = mouseY - clickOffset.y;
    }
}
```

Observez le fichier **DraggingSprites.fla** pour voir comment ce code fonctionne. Essayez de faire glisser la mascotte de différents points afin de voir comment `clickOffset` gère ces différences.

Détection de collisions

Une fois que vos objets se déplaceront à l'écran dans votre jeu, il arrivera très couramment que vous deviez vérifier s'ils entrent en collision les uns avec les autres.

ActionScript 3.0 contient deux fonctions de détection de collision natives. La fonction `hitTestPoint` teste un emplacement de point afin de voir s'il se trouve à l'intérieur d'un objet d'affichage. La fonction `hitTestObject` compare deux objets d'affichage l'un à l'autre afin de voir s'ils se chevauchent.

Pour examiner ces deux fonctions, créons un exemple simple qui examine l'emplacement du curseur et l'emplacement d'un sprite qui se déplace à chaque image :

```
addEventListener(Event.ENTER_FRAME, checkCollision);
```

La fonction `checkCollision` commence par utiliser `hitTestPoint` en recherchant l'emplacement du curseur afin de voir s'il touche le clip du croissant dans la scène. Les deux premiers paramètres de la fonction `hitTestPoint` sont l'emplacement `x` et `y` du point. Le troisième paramètre correspond au type de limite à utiliser. La valeur par défaut, `false`, signifie que seul le rectangle de contour de l'objet d'affichage doit être pris en compte.

À moins que le sprite ne possède la forme d'un rectangle, cela ne suffit pas pour la plupart des usages pratiques dans les jeux. En positionnant le troisième paramètre à `true`, `hitTestPoint` utilisera cette fois la forme effective de l'objet d'affichage afin de déterminer la collision.

Nous placerons un texte différent dans un champ texte de message selon le résultat de `hitTestPoint` :

```
function checkCollision(event:Event) {

    // Vérifier l'emplacement du curseur par rapport au croissant
    if (crescent.hitTestPoint(mouseX, mouseY, true)) {
        messageText1.text = "hitTestPoint: YES";
    } else {
        messageText1.text = "hitTestPoint: NO";
    }
```

La fonction `hitTestObject` ne propose pas d'option de forme. Elle ne fait que comparer les deux rectangles de contour des deux sprites. Elle peut cependant être utile dans certains cas.

Le fragment de code suivant amène un clip d'étoile à suivre le curseur et place un message différent dans un autre champ texte si les rectangles de contour entrent en intersection :

```
    // Déplacer l'étoile avec la souris
    star.x = mouseX;
    star.y = mouseY;

    // Vérifier si l'étoile touche le croissant
    if (star.hitTestObject(crescent)) {
        messageText2.text = "hitTestObject: YES";
    } else {
        messageText2.text = "hitTestObject: NO";
    }
}
```

L'animation d'exemple **CollisionDetection.fla** illustre cet exemple. La Figure 2.17 montre que le curseur se trouve à l'intérieur du rectangle de contour du croissant ; comme nous testons `hitTestPoint` avec le drapeau de forme positionné à `true`, aucune collision n'est enregistrée à moins que le curseur ne se trouve effectivement au-dessus du croissant. L'étoile et le croissant, pendant ce temps, entrent en collision lorsque leurs rectangles de contour entrent en intersection.

Figure 2.17
Les emplacements du curseur et de l'étoile sont testés afin de vérifier s'ils entrent en collision avec le croissant.

Accéder à des données externes

Parfois, il peut être nécessaire d'accéder à des informations situées hors du jeu. Vous pouvez charger des paramètres de jeu externes depuis des pages Web ou des champs texte. Vous pouvez également enregistrer et charger des informations localement.

Variables externes

Supposons que vous ayez un jeu susceptible de varier en fonction de certaines options, comme un jeu de puzzle qui pourrait utiliser différentes images ou un jeu d'arcade qui pourrait s'exécuter à différentes vitesses.

Vous pouvez alimenter les valeurs des variables dans l'animation Flash à partir de la page HTML dans laquelle elle se trouve.

Il existe plusieurs moyens de s'y prendre. Si vous utilisez le modèle HTML par défaut des paramètres de publication, vous pouvez passer des valeurs de paramètre *via* la propriété `flashvars` de la fonction `AC_FL_RunContent`.

*L'animation Flash est incorporée dans la page Web à l'aide des balises `OBJECT` et `EMBED` pour les architectures ActiveX (Internet Explorer) et Plug-In (Firefox). Les balises `OBJECT` et `EMBED` sont cependant à leur tour écrites par un morceau de JavaScript fourni par Adobe et livré dans le fichier **AC_RunActiveContent.js**. Selon vos paramètres de publication, vous obtiendrez une copie de ce fichier à chaque publication.*

Voici une version raccourcie de l'appel à `AC_FL_RunContent` que vous trouverez dans le fichier HTML exporté lors de la publication depuis Flash CS3. Il inclut un paramètre `flashvars` ajouté par mes soins :

```
<script language="JavaScript">
AC_FL_RunContent(
    'codebase', 'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,0,0',
    'width', '550',
    'height', '400',
    'src', 'ExternalVariables',
    'quality', 'high',
    'flashvars', 'puzzleFile=myfilename.jpg&difficultyLevel=7'
    );
</script>
```

Le format flashvars est une série de paires *nom de propriété = valeur* séparées par le symbole &. Dans cet exemple, la propriété puzzleFile se voit donc attribuer la valeur myfilename.jpg, tandis que la propriété difficultyLevel reçoit la valeur 7.

Lorsque l'animation Flash démarre, elle peut obtenir ces valeurs en utilisant l'objet LoaderInfo. La ligne suivante récupère tous les paramètres et les place dans un objet :

```
var paramObj:Object = LoaderInfo(this.root.loaderInfo).parameters;
```

Pour accéder à une valeur individuelle, il vous suffit d'utiliser une ligne de code de ce genre :

```
var diffLevel:String = paramObj["difficultyLevel"];
```

Vous pouvez passer n'importe quel nombre de constantes de jeu, comme des noms d'image, des niveaux de départ, des vitesses, des positions, etc. Un jeu de pendu peut ainsi être configuré avec un autre mot ou une autre phrase. Un jeu d'exploration terrestre peut se voir attribuer un autre emplacement pour le point de départ.

Lors de l'exécution de l'animation **ExternalVariables.fla**, gardez à l'esprit que le principe consiste à charger la page **ExternalVariables.html** dans votre navigateur. Cette page contient tous les paramètres flashvars définis. Si vous essayez d'effectuer votre test dans Flash ou de créer une nouvelle page HTML, ces paramètres manqueront.

Charger des données

Le chargement de données depuis un fichier texte externe est relativement facile. S'il s'agit d'un fichier au format XML, la procédure s'effectue même de manière idéale.

Par exemple, supposons que vous souhaitiez charger une question dans un quiz à partir d'un fichier. Les données XML pourraient ressembler à ceci :

```
<LoadingData>
    <question>
        <text>This is a test</text>
        <answers>
            <answer type="correct">Correct answer</answer>
            <answer type="wrong">Incorrect answer</answer>
        </answers>
    </question>
</LoadingData>
```

Pour charger les données, vous devez utiliser deux objets : un URLRequest et un URLLoader. Ensuite, vous effectuez une écoute pour vérifier que le chargement est complet et vous appelez l'une de vos propres fonctions :

```
var xmlURL:URLRequest = new URLRequest("LoadingData.xml");
var xmlLoader:URLLoader = new URLLoader(xmlURL);
xmlLoader.addEventListener(Event.COMPLETE, xmlLoaded);
```

Le `xmlLoaded`, dans le cas présent, correspond simplement à des instructions `trace` afin de montrer que des données ont été importées :

```
function xmlLoaded(event:Event) {
    var dataXML  = XML(event.target.data);
    trace(dataXML.question.text);
    trace(dataXML.question.answers.answer[0]);
    trace(dataXML.question.answers.answer[0].@type);
}
```

Vous voyez à quel point il est facile de récupérer les données XML du fichier. L'objet XML étant `dataXML`, vous pouvez récupérer le texte de la question avec `dataXML.question.text` et la première réponse avec `dataXML.question.answers[0]`. Vous pouvez récupérer un attribut, comme le type de la réponse, en utilisant `@type`.

L'exemple **LoadingData.fla** lit ses données à partir du fichier **LoadingData.xml**. Essayez de modifier et d'ajouter des données au fichier XML. Ensuite, lancez l'animation avec les instructions `trace` afin d'accéder aux différentes parties des données.

Enregistrer des données locales

L'un des besoins courants en matière de développement de jeu consiste à stocker des données locales. Par exemple, vous pourriez stocker le précédent score du joueur ou certaines options de votre jeu.

Pour stocker des données sur l'ordinateur de l'utilisateur, nous allons utiliser un objet `SharedObject` local. L'accès à un `SharedObject` s'effectue par la même opération que sa création. Le fait de demander s'il existe le crée.

Pour cela, attribuez une variable au `SharedObject` d'un certain nom, avec la fonction `getLocal` :

```
var myLocalData:SharedObject = SharedObject.getLocal("mygamedata");
```

L'objet `myLocalData` peut prendre n'importe quel nombre de propriétés de n'importe quel type : nombres, chaînes, tableaux, autres objets, etc.

Si vous aviez stocké les mêmes données dans une propriété de l'objet partagé nommé `gameinfo`, vous pourriez y accéder avec `myLocalData.data.gameinfo` :

```
trace("Found Data: "+myLocalData.data.gameinfo);
```

Définissez donc cette propriété `gameinfo`, comme vous le feriez pour une variable standard :

```
myLocalData.data.gameinfo = "Store this.";
```

Essayez d'exécuter l'animation test **SavingLocalData.fla**. Elle utilise la fonction `trace` pour afficher la propriété `myLocalData.data.gameinfo`. Comme elle n'a pas été définie, vous obtenez le résultat `undefined`. Ensuite, elle positionne la valeur. Lors de la seconde exécution du test, vous obtenez ainsi `"Store this."`.

Éléments de jeu divers

Voici quelques scripts simples qui réalisent différentes tâches. La plupart d'entre eux peuvent être ajoutés à n'importe quelle animation de jeu en cas de besoin.

Curseurs personnalisés

Supposons que vous souhaitiez remplacer le curseur de souris standard par un curseur qui correspond mieux au style de votre jeu. Ou supposons encore que vous souhaitiez un curseur plus grand pour un jeu d'enfant ou un curseur en forme de cible pour un jeu de tir.

Si vous ne pouvez pas modifier le curseur de l'ordinateur, vous pouvez le faire disparaître, à tout le moins visuellement. Ensuite, vous pouvez le remplacer par un sprite dont la position correspond à celle du curseur et qui flotte au-dessus de tous les autres éléments.

Pour rendre le curseur invisible, utilisez la commande `Mouse.hide()` :

```
Mouse.hide();
```

Ensuite, pour faire agir un sprite en tant que curseur, placez-le dans un calque au-dessus de tous les autres éléments de l'écran. La Figure 2.18 présente le scénario avec trois claques. Le curseur est le seul élément du second calque et tous les autres éléments se trouvent en dessous dans le troisième calque.

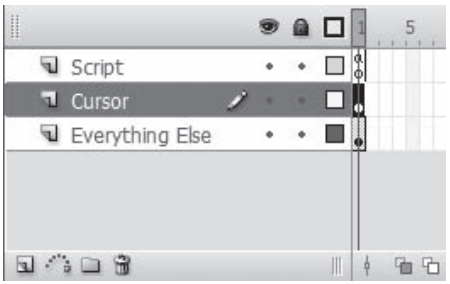


Figure 2.18
Le curseur doit rester au-dessus de tous les autres éléments à l'écran.

Si vous créez des objets avec ActionScript, vous devez veiller à conserver le curseur au-dessus de tous les objets. La commande `setChildIndex` vous permet ainsi de placer le curseur en haut après avoir créé les objets de votre jeu.

Pour amener un sprite à suivre le curseur, il nous faut un écouteur `ENTER FRAME` :

```
addEventListener(Event.ENTER_FRAME, moveCursor);
```

Ensuite, la commande `moveCursor` amène simplement l'objet `arrow`, qui correspond ici au nom d'instance du curseur dans la scène, à suivre l'emplacement de la souris :

```
function moveCursor(event:Event) {
    arrow.x = mouseX;
    arrow.y = mouseY;
}
```

Vous devez également positionner la propriété `mouseEnabled` du sprite à `false`. Sans cela, le curseur masqué se trouverait toujours au-dessus du sprite du curseur et jamais au-dessus des sprites qui se trouvent en dessous de lui, comme un bouton :

```
arrow.mouseEnabled = false;
```

Sans cette ligne de code, votre bouton ne fait pas apparaître son état de survol lorsque vous le survolez et ne réceptionne pas correctement les clics de souris. Cette ligne de code rend le curseur personnalisé invisible pour les événements de souris.

La Figure 2.19 présente le curseur personnalisé qui survole un bouton.

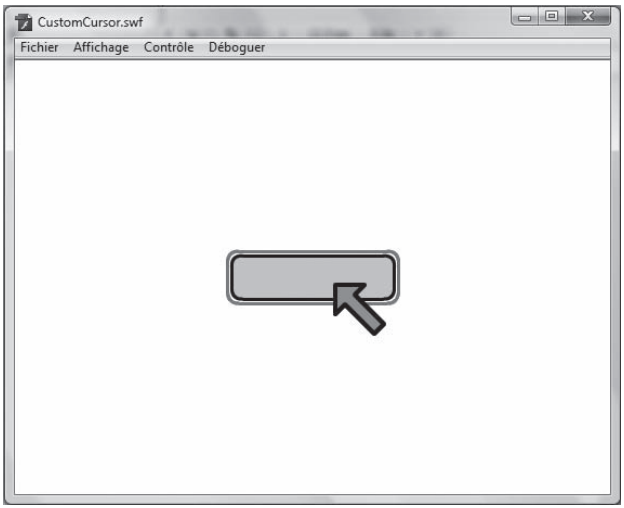


Figure 2.19
Le bouton présente son état de survol bien que le sprite `arrow` soit techniquement le premier sprite sous l'emplacement de la souris.

L'animation d'exemple **CustomCursor.fla** contient un bouton simple dans la scène afin que vous puissiez tester le survol du curseur personnalisé sur un bouton.

Lire des sons

Il existe deux principaux moyens de lire des sons en ActionScript 3.0 : en tant que sons de bibliothèque interne ou en tant que fichiers externes.

La meilleure méthode pour la plupart des effets de son de jeu consiste à incorporer les sons dans la bibliothèque de l'animation du jeu.

Vous pouvez le faire en important le son avec la commande de menu Fichier > Importer > Importer dans la bibliothèque. Une fois que le son est dans la bibliothèque, sélectionnez-le et examinez sa boîte de dialogue Propriétés audio (voir Figure 2.20).

Pour utiliser un son dans votre code ActionScript, vous devez définir la liaison du son à exporter pour ActionScript, puis attribuer à la classe un nom que vous allez utiliser dans votre code. Pour cet exemple, nous utiliserons le nom `Sound1`.

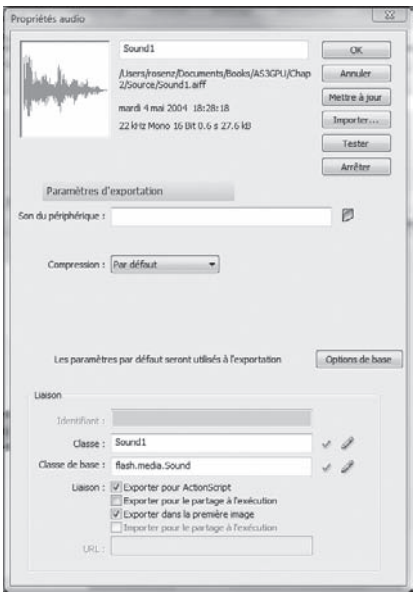
Pour lire le son, vous n'avez dès lors besoin que de deux lignes de code :

```
var sound1:Sound1 = new Sound1();
var channel:SoundChannel = sound1.play();
```

Si vous souhaitez être plus concis, il est même possible de procéder en une seule ligne :

```
var channel:SoundChannel = (new Sound1()).play();
```

Figure 2.20
La boîte de dialogue Propriétés audio permet de définir l'identificateur de classe pour un son afin de pouvoir l'utiliser dans le code ActionScript.



La lecture d'un fichier externe est légèrement plus difficile. Pour commencer, vous devez charger le son dans un objet. Le code suivant charge le fichier son **PlayingSounds.mp3** dans l'objet `sound2` :

```
var sound2:Sound = new Sound();
var req:URLRequest = new URLRequest("PlayingSounds.mp3");
sound2.load(req);
```

Ensuite, pour lire le son, vous devez utiliser la commande `play` :

```
sound2.play();
```

L'animation d'exemple **PlayingSounds.fla** contient deux boutons : l'un qui lit un son de bibliothèque et l'autre qui lit un son externe. Le son externe est chargé dès que l'animation commence et se trouve donc prêt à être lu à tout moment.

Avec certains sons externes plus longs, il est possible que le son n'ait pas fini de se charger avant qu'il ne soit requis. Vous pouvez détecter ce cas en utilisant la propriété `isBuffering` de l'objet son. Vous pouvez également utiliser les propriétés `bytesLoaded` et `bytesTotal` pour un suivi plus précis.

Si le son n'a pas fini de se charger, il commencera cependant à jouer aussitôt qu'il le sera. Pour les sons courts, il n'est donc probablement pas utile de se préoccuper à ce sujet.

Écran de chargement

Flash est conçu pour une diffusion du contenu en flux continu. Cela signifie que l'animation ne commence que lorsque le contenu minimal requis a bien été chargé, comme les éléments utilisés pour la première image.

Ce mécanisme convient à merveille pour les animations. Vous pouvez proposer une animation cousue main de 1 000 images qui démarre immédiatement et continue à charger les éléments requis pour les prochaines images à mesure que l'utilisateur en observe les précédentes.

Pour les jeux, il est cependant rare que l'on procède de cette manière. Les éléments de jeu sont utilisés presque immédiatement par le code ActionScript. Si l'un d'entre eux venait à manquer parce qu'il n'était pas chargé, le jeu pourrait ne pas fonctionner correctement.

La plupart des jeux utilisent donc un écran de chargement qui force l'animation à attendre que tous les éléments aient d'abord été téléchargés. Cet écran contribue aussi à tenir le joueur informé de l'état du téléchargement.

L'un des moyens simples de créer un écran de ce type consiste à insérer une instruction `stop` dans la première image de l'animation afin que celle-ci ne commence sa lecture qu'une fois que vous le lui aurez indiqué spécifiquement :

```
stop();
```

Ensuite, définissez un écouteur `ENTER_FRAME` pour appeler une fonction `loadProgress` à chaque image :

```
addEventListener(Event.ENTER_FRAME, loadProgress);
```

Cette fonction récupère l'état de l'animation en utilisant `this.root.loaderInfo`. Elle a des propriétés `bytesLoaded` et `bytesTotal`. Nous les prendrons et les convertirons également en kilo-octets grâce à une division par 1 024 :

```
function loadProgress(event:Event) {
    // Récupérer les octets chargés et le nombre total d'octets
    var movieBytesLoaded:int = this.root.loaderInfo.bytesLoaded;
    var movieBytesTotal:int = this.root.loaderInfo.bytesTotal;

    // Conversion en kilo-octets
    var movieKLoaded:int = movieBytesLoaded/1024;
    var movieKTotal:int = movieBytesTotal/1024;
```

Pour indiquer au joueur la progression du chargement, nous plaçons du texte dans un champ texte qui se trouve déjà dans l'image 1 de l'animation. Le message donnera une information du type `"Loading: 5K/32K"` :

```
    // Afficher la progression
    progressText.text = "Loading: "+movieKLoaded+"K/"+movieKTotal+"K";
```

Lorsque `movieBytesLoaded` atteint `movieBytesTotal`, nous supprimons l'écouteur événementiel et conduisons l'animation à l'image 2. S'il s'agit du début d'une séquence animée, vous pouvez utiliser `gotoAndPlay` à la place :

```
    // Avancer si OK
    if (movieBytesLoaded >= movieBytesTotal) {
        removeEventListener(Event.ENTER_FRAME, loadProgress);
        gotoAndStop(2);
    }
}
```

L'animation d'exemple **LoadingScreen.fla** contient ce code dans la première image. Elle contient aussi une image de 33 Ko dans la seconde image. Pour tester ce code, commencez par tester l'animation normalement en choisissant Contrôle > Tester l'animation. Ensuite, dans l'environnement de test, choisissez Affichage > Simuler le téléchargement. Ce réglage simule un téléchargement à 4,7 Ko/seconde et permet de voir l'écran de chargement en action.

Nombres aléatoires

Les nombres aléatoires sont utilisés dans presque tous les jeux. Ils permettent de réaliser des variations à l'infini et contribuent à simplifier votre code.

En ActionScript 3.0, la création de nombres aléatoires s'opère avec la fonction `Math.random`. Elle retourne une valeur comprise entre 0,0 et 1,0 sans inclure 1,0 lui-même.

Le code suivant récupère ainsi un nombre compris entre 0,0 et 1,0 sans inclure 1,0 :

```
var random1:Number = Math.random();
```

Le nombre retourné est généré par un algorithme complexe dans le lecteur Flash. Il semble être complètement aléatoire. Pourtant, étant donné qu'il s'agit d'un algorithme, il n'est techniquement pas complètement aléatoire. Pour nos besoins en matière de développement de jeux, nous n'aurons cependant pas à nous en soucier et pourrons considérer que les nombres retournés sont complètement aléatoires.

En général, vous souhaiterez définir une plage plus spécifique pour le nombre aléatoire. Par exemple, vous pourriez souhaiter un nombre aléatoire compris entre 0 et 10. Pour définir ces plages, il suffit de multiplier le résultat de `Math.random` par la plage concernée :

```
var random2:Number = Math.random()*10;
```

Si vous souhaitez une valeur entière au lieu d'un nombre à virgule flottante, utilisez `Math.floor` pour arrondir les valeurs à l'entier inférieur. Le code suivant fournit un nombre aléatoire compris entre 0 et 9 :

```
var random3:Number = Math.floor(Math.random()*10);
```

Si vous souhaitez définir une plage qui ne commence pas à 0, ajoutez la valeur requise au résultat. Le code suivant donne un nombre aléatoire compris entre 1 et 10 :

```
var random4:Number = Math.floor(Math.random()*10)+1;
```

L'animation **RandomNumbers.fla** présente ces lignes de code avec une sortie dans le panneau Sortie.

Mélanger un tableau

L'un des usages les plus courants des nombres aléatoires dans les jeux consiste à configurer les pièces du jeu au début d'une partie. En général, cela implique un mélange des éléments du jeu, comme des cartes ou des pièces de jeu.

Par exemple, supposons que vous ayez cinquante-deux pièces de jeu à mélanger dans un ordre aléatoire, comme un croupier mélangerait des cartes avant de servir une main de poker ou de blackjack.

L'opération consiste à créer d'abord le tableau des pièces de jeu sous forme de simple tableau trié. Le code qui suit le fait avec les nombres 0 à 51 :

```
// Créer un tableau trié
var startDeck:Array = new Array();
for(var cardNum:int=0;cardNum<52;cardNum++) {
    startDeck.push(cardNum);
}
trace("Unshuffled:",startDeck);
```

Le résultat dans la fenêtre Sortie ressemble à ceci :

```
Unshuffled: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28
,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51
```

Pour mélanger le tableau de manière aléatoire, nous choisissons une position aléatoire à l'intérieur et prenons le nombre de cette position pour le placer dans un nouveau tableau. Ensuite, nous supprimons le nombre de l'ancien tableau. Nous poursuivons ainsi de manière répétitive jusqu'à ce que l'ancien tableau soit vidé :

```
// Mélange dans un nouveau tableau
var shuffledDeck:Array = new Array();
while (startDeck.length > 0) {
    var r:int = Math.floor(Math.random()*startDeck.length);
    shuffledDeck.push(startDeck[r]);
    startDeck.splice(r,1);
}
trace("Shuffled:", shuffledDeck);
```

Le résultat ressemblera à ceci (il sera évidemment différent à chaque fois que vous exécuterez le programme) :

```
Shuffled: 3,42,40,16,41,44,30,27,33,11,50,0,21,23,49,29,20,28,22,32,39,25,17,19,8,7,10,3
7,2,12,31,5,46,26,48,45,43,9,4,38,15,36,51,24,14,18,35,1,6,34,13,47
```

L'animation d'exemple **ShufflingAnArray.fla** présente cette procédure.

Afficher une horloge

Il est possible d'obtenir une mesure du temps courant avec la fonction `getTimer()`. Cette fonction indique le nombre de millisecondes qui se sont écoulées depuis le démarrage du lecteur Flash.

En général, les horloges internes des jeux notent le début du jeu en plaçant la valeur `getTimer()` à cet instant dans une variable. Par exemple, le jeu peut commencer 7,8 secondes après le démarrage du lecteur Flash s'il a fallu ce temps à l'utilisateur pour trouver le bouton "Jouer" et cliquer dessus. La valeur 7800 est alors stockée dans `startTime`.

Ensuite, pour obtenir la mesure temporelle à tout moment, il suffit de soustraire `startTime` de la mesure de temps actuelle.

Le joueur ne s'intéressera cependant que rarement à des millisecondes brutes. Il souhaitera plutôt voir quelque chose comme "1:34" pour 1 minute et 34 secondes.

La conversion des millisecondes en un format d'horloge requiert simplement une division par 1 000 pour obtenir le nombre de secondes, puis par 60 pour obtenir le nombre de minutes.

Voici un exemple de programme qui place un champ texte à l'écran, capture la mesure temporelle au début puis affiche l'horloge à chaque image. Il convertit la mesure temporelle en secondes et en minutes, en insérant un zéro dans le nombre de secondes s'il est inférieur à dix :

```
var timeDisplay:TextField = new TextField();
addChild(timeDisplay);

var startTime:int = getTimer();
addEventListener(Event.ENTER_FRAME, showClock);

function showClock(event:Event) {
    // Millisecondes écoulées
    var timePassed:int = getTimer()-startTime;

    // Calcul des minutes et des secondes
    var seconds:int = Math.floor(timePassed/1000);
    var minutes:int = Math.floor(seconds/60);
    seconds -= minutes*60;

    // Création de la chaîne d'horloge
    var timeString:String = minutes+":"+String(seconds+100).substr(1,2);

    // Affichage dans le champ texte
    timeDisplay.text = timeString;
}
```

Examinons de plus près la conversion de chaîne. Le nombre de minutes est récupéré directement à partir de la variable `minutes`. Le signe deux-points est ajouté ensuite.

Le nombre de secondes est géré différemment : 100 lui est ajouté, de sorte que 7 secondes devient 107 secondes ; 52 devient 152 secondes, etc. Ensuite, il est converti en une chaîne avec le constructeur `String`. Nous récupérons alors la sous-chaîne démarrant au caractère 1 et de longueur 2. Comme nous commençons à compter les caractères à partir de 0, cela signifie que nous obtenons 07 ou 52, sans inclure le 1 au début de 107 ou de 152.

Le résultat donne une chaîne comme `1:07` ou `23:52`. Consultez l'animation d'exemple **DisplayingAClock.fla** pour voir ce code en action.

Données système

Il peut souvent être nécessaire d'obtenir des informations concernant le type d'ordinateur sur lequel votre jeu est exécuté. Ces indications pourraient affecter la manière dont vous souhaitez que votre jeu gère différentes situations ou le niveau de détail que vous désirez proposer.

Par exemple, vous pouvez récupérer la largeur et la hauteur de la scène avec les propriétés `stage.stageWidth` et `stage.stageHeight`. Ces valeurs changent même en temps réel si l'animation est configurée pour s'ajuster à la taille de la fenêtre du navigateur.

Si votre animation est conçue pour faire 640 pixels de large et que vous détectiez qu'elle se lit sur 800 pixels de largeur, vous pouvez choisir d'afficher plus de détails afin que le joueur profite de cette précision renforcée. Vous pouvez aussi choisir à l'inverse d'afficher moins d'images d'animation, car plus l'échelle est grande plus il faut de puissance pour le rendu.

Vous pouvez encore utiliser l'objet `Capabilities` pour obtenir différents éléments d'information concernant l'ordinateur. Voici une liste pratique des éléments qui vous affecteront le plus en tant que développeur de jeux :

- **`Capabilities.playerType`.** Retourne `External` si vous testez l'animation, `StandAlone` si elle s'exécute sous forme de projecteur Flash, `PlugIn` si elle s'exécute dans un navigateur comme FireFox ou Safari ou `ActiveX` si elle s'exécute dans Internet Explorer. Vous pouvez donc insérer dans votre code des fragments qui ne fonctionnent que si `playerType` vaut `External` et vous permettent de tester votre jeu sans affecter la version Web.
- **`Capabilities.language`.** Retourne le code à deux lettres, comme `en` pour l'anglais, si l'ordinateur est configuré de manière à utiliser cette langue comme langue principale.
- **`Capabilities.os`.** Retourne le type et la version du système d'exploitation, comme `Mac OS 10.4.9`.
- **`Capabilities.screenResolutionX`**, **`Capabilities.screenResolutionY`**. La résolution d'affichage, comme 1280 et 1024.
- **`Capabilities.version`**. La version du lecteur Flash, comme `MAC 9,0,45,0`. Vous pouvez extraire la version du système d'exploitation ou celle du lecteur de ce code.

Bien d'autres propriétés `Capabilities` sont disponibles. Consultez la documentation Flash CS3 à ce sujet. Examinez le fichier **SystemData.fla** pour un exemple d'animation qui récupère la plupart des données précédentes et les affiche directement dans un champ texte.

Sécurité et vol des jeux

Le vol de jeu est un véritable problème sur Internet. La plupart des jeux ne sont pas du tout protégés et rien n'empêche le premier venu de récupérer le fichier SWF et de le télécharger sur son site Web en prétendant qu'il est le fruit de son propre labeur.

Il existe de nombreux moyens d'empêcher ces agissements. Le plus simple consiste à amener votre jeu à s'assurer qu'il s'exécute réellement depuis votre serveur. Cette vérification peut s'opérer avec la propriété `this.root.loaderInfo.url`. Celle-ci retourne le chemin complet du fichier en commençant par `http://` si le fichier se trouve sur le Web.

Vous pouvez ensuite opérer une vérification portant sur le domaine. Par exemple, pour vous assurer que `flashgameu.com` apparaît dans le chemin, procédez de la manière suivante :

```
if (this.root.loaderInfo.url.indexOf("flashgameu.com") != -1) {
    info.text = "Is playing at flashgameu.com";
} else {
    info.text = "Is NOT playing at flashgameu.com";
}
```

Au lieu de définir simplement un champ texte, vous pouvez arrêter la lecture du jeu ou conduire le joueur à votre site avec `navigateToURL`.

Une fois que vous avez sécurisé votre jeu au niveau de votre site, l'étape suivante consiste à le sécuriser de manière que personne ne puisse utiliser une balise `EMBED` avec une URL absolue vers votre fichier SWF. Cette méthode permet en effet aux voleurs d'incorporer votre jeu depuis votre serveur directement dans leur page Web sur leur serveur.

Il n'existe aucun moyen facile d'empêcher cela mais, si vous découvrez ce subterfuge, vous pouvez toujours déplacer votre fichier SWF. Vous pouvez remplacer votre fichier SWF par un autre fichier qui se contente de rediriger le joueur avec `navigateToURL`.

Certains serveurs Web peuvent empêcher la liaison distante. Le but est avant tout d'empêcher les utilisateurs d'incorporer des images de votre serveur dans leurs pages Web. Dans de nombreux cas, cela fonctionne aussi avec les fichiers SWF. Renseignez-vous auprès de votre FAI concernant cette fonctionnalité.

Il existe une autre méthode plus avancée et relativement complexe pour empêcher la liaison incorporée. Elle implique de passer une valeur secrète à l'animation Flash par deux biais : sous forme de paramètre `flashvars` et sous la forme d'un fragment de texte ou de données XML avec `URLLoader`. Si les deux valeurs secrètes ne se correspondent pas, on en déduit que l'animation Flash a dû être volée.

L'idée est de modifier régulièrement la valeur passée par les deux biais. Si quelqu'un vole votre animation Flash mais ne récupère pas votre code HTML pour incorporer l'animation Flash dans la page, votre animation ne récupère dès lors pas la version `flashvars` de la valeur secrète et ne peut fonctionner pour cette personne.

Si la personne vole votre code HTML, elle ne possède que la version courante de la valeur secrète `flashvars`. Pour l'instant, cette valeur correspond à la valeur secrète `URLLoader` mais, une fois que vous aurez mis à jour la valeur secrète aux deux endroits, l'ancienne valeur `flashvars` dans la page du voleur cessera de correspondre à la nouvelle valeur `URLLoader` de votre serveur.

Il reste évidemment possible que le pirate vole votre jeu SWF, l'ouvre avec un décompilateur SWF et supprime le code de sécurité. Il n'existe pas de solution sécurisée à 100 %. La plupart des voleurs recherchent cependant des jeux faciles à voler. Le vôtre n'en fera ainsi pas partie.

Maintenant que vous avez découvert quelques-unes des techniques de programmation ActionScript 3.0 grâce à ces petits blocs constructeurs de code, il est temps de passer à la réalisation de votre premier jeu.

3

Structure de jeu élémentaire : le Memory

Au sommaire de ce chapitre :

- Placer des éléments interactifs
- Jeu
- Encapsuler un jeu
- Ajouter un score et un chronomètre
- Ajouter des effets de jeu
- Modifier le jeu

Codes sources

http://flashgameu.com

A3GPU03_MatchingGame.zip

Pour créer votre premier jeu, j'ai choisi l'un des jeux les plus populaires que vous puissiez trouver sur le Web et dans l'univers du logiciel interactif et éducatif : le Memory.

Le Memory est un simple jeu de correspondance qui se joue habituellement avec un jeu de cartes représentant des images. L'idée consiste à placer des paires de cartes face cachée selon une disposition aléatoire. Ensuite, le ou les joueurs doivent tenter de trouver des correspondances en retournant deux cartes de suite. Lorsque deux cartes se correspondent, elles sont retirées. Si elles ne se correspondent pas, elles sont retournées (et donc cachées) de nouveau.

Les bons joueurs sont ceux qui mémorisent les cartes aperçues lorsque les cartes ne se correspondent pas et qui parviennent à déterminer où se trouvent les vraies paires après plusieurs tentatives manquées.

Certains des jeux de correspondance éducatifs pour enfants proposent non pas des correspondances exactes pour les paires de cartes mais des jeux d'association. Par exemple, une carte peut contenir une image de chat et sa carte correspondante, contenir le mot Chat. Une carte peut afficher le numéro 7 et l'autre, la somme 3+4.

Les versions informatiques des jeux de correspondance possèdent plusieurs avantages sur leurs équivalents physiques : vous n'avez pas besoin de récupérer, de mélanger et de placer les cartes au début de chaque jeu. L'ordinateur s'occupe de tout cela pour vous. Il est également plus facile et moins coûteux pour le développeur du jeu de créer différentes images pour les cartes avec des cartes virtuelles qu'avec de vraies cartes !

Pour créer un jeu de Memory, nous devons d'abord placer des cartes à l'écran. Pour cela, nous devons mélanger le jeu afin que les cartes soient disposées dans un ordre aléatoire à chaque nouvelle partie.

Ensuite, nous devons récupérer l'entrée de l'utilisateur et nous en servir pour révéler les images de deux cartes sélectionnées. Nous devons encore comparer les cartes et les retirer en cas de correspondance.

Nous devons enfin retourner les cartes pour les cacher de nouveau lorsqu'elles ne se correspondent pas et vérifier à quel moment toutes les paires ont été trouvées pour que la partie puisse se terminer.

Placer les éléments interactifs

Pour créer un jeu de Memory, nous devons d'abord créer un jeu de cartes. Ces cartes devant aller par paires, il convient de déterminer quel nombre sera affiché à l'écran et de créer la moitié de ce nombre d'images.

Par exemple, si nous souhaitons afficher trente-six cartes dans le jeu, il nous faut dix-huit images, qui apparaîtront chacune sur deux cartes.

Méthodes pour la création des pièces du jeu

Il existe deux écoles en matière de création de pièces de jeu et notamment pour la création de cartes dans un jeu de Memory.

Méthode à symboles multiples

La première méthode consiste à créer chaque carte sous forme de clip individuel. Dans ce cas, il y aura dix-huit symboles. Chacun représente une carte.

L'un des problèmes avec cette méthode tient à ce qu'il y a de fortes chances pour que vous deviez dupliquer certains éléments graphiques à l'intérieur de chacun des symboles. Par exemple, chaque carte doit posséder la même bordure et le motif de dos. Vous aurez ainsi dix-huit copies de la bordure et du dos.

Vous pouvez évidemment éviter cela en créant un symbole de dos de carte et en l'utilisant dans chacun des dix-huit symboles de carte.

Le recours aux symboles multiples peut être utile si vous récupérez des cartes dans un groupe de grande taille – par exemple si vous avez besoin de dix-huit cartes sur un jeu de cent cartes. Cette méthode peut aussi être utile si les cartes sont importées dans l'animation à partir de fichiers multimédias externes, comme une série d'images JPG.

La méthode à plusieurs symboles présente cependant des inconvénients lorsqu'il s'agit d'opérer des changements. Par exemple, supposons que vous souhaitiez redimensionner légèrement les images. Vous devrez alors le faire dix-huit fois pour dix-huit symboles différents.

En outre, si vous associez vos talents de programmeur à ceux d'un graphiste, il n'est pas souhaitable que le graphiste ait à mettre à jour dix-huit symboles ou plus. S'il est sous contrat, vous risquez d'engloutir rapidement tout votre budget !

Méthode à symbole unique

La seconde méthode pour travailler avec une série de pièces de jeu comme des cartes est la méthode à symbole unique. Vous utilisez dans ce cas un symbole (un clip) avec plusieurs images. Chaque image contient le graphisme d'une carte différente. Les graphismes partagés, comme la bordure et l'arrière-plan, peuvent alors se trouver sur un calque du clip qui s'étend sur toutes les images.

Cette méthode possède d'indéniables avantages lorsqu'il s'agit de mettre à jour et de modifier les pièces du jeu. Vous pouvez aisément et rapidement vous déplacer entre les images et les éditer dans le clip. Vous pouvez également récupérer facilement un clip mis à jour par un graphiste avec lequel vous travaillez.

La méthode à symbole unique peut elle aussi utiliser plusieurs symboles. Par exemple, si vos pièces de jeu correspondent à un jeu de cartes de poker, vous pouvez placer les quatre suites (pique, cœur, carreau et trèfle) dans des symboles et les utiliser dans le symbole principal de votre jeu de cartes. Ainsi, si vous souhaitez modifier l'apparence du cœur dans le jeu de cartes complet, vous pouvez le faire en ne modifiant que le symbole de cœur.

Configurer l'animation Flash

Grâce à la méthode à symbole unique, il nous faut au moins un clip dans la bibliothèque. Ce clip contiendra toutes les cartes et même une image qui représente le dos de la carte que nous devons afficher lorsque la carte est retournée face en bas.

Créez une nouvelle animation qui contient un unique clip appelé `Card`. Pour créer une nouvelle animation dans Flash CS3, choisissez Fichier > Nouveau. Une liste de types de fichiers apparaît. Choisissez Fichier Flash (ActionScript 3.0) pour créer un fichier d'animation qui fonctionne avec le fichier de classe ActionScript 3.0 que nous sommes sur le point de créer.

Placez au moins dix-neuf images dans ce clip, l'une représentant le dos des cartes et les dix-huit autres représentant les faces avec différentes images. Ouvrez le fichier **MatchingGame1.fla** pour cet exercice si vous n'avez pas votre propre fichier de symboles à utiliser.

La Figure 3.1 présente un scénario pour le clip `Card` que nous utiliserons dans ce jeu. La première image correspond à l'arrière des cartes. C'est ce que le joueur verra lorsque la carte sera tournée face en bas. Ensuite, chacune des autres images présente une image différente pour la face d'une carte.

Une fois qu'un symbole se trouve dans la bibliothèque, il faut le configurer pour pouvoir l'utiliser dans notre code ActionScript. Pour cela, nous devons définir ses propriétés en le sélectionnant dans la bibliothèque et en affichant la boîte de dialogue Propriétés du symbole (voir Figure 3.2).

Tapez le nom de symbole Card et choisissez le type Clip. Pour qu'ActionScript puisse opérer avec le clip Cards, il faut que ce dernier se voie attribuer une classe. En cochant l'option Exporter pour ActionScript, nous attribuons automatiquement le nom de classe Card au symbole. Cela conviendra parfaitement à nos besoins.

Rien d'autre n'est requis dans l'animation Flash. Le scénario principal est entièrement vide. La bibliothèque ne contient qu'un seul clip : Card. Il ne nous manque plus que du code ActionScript.

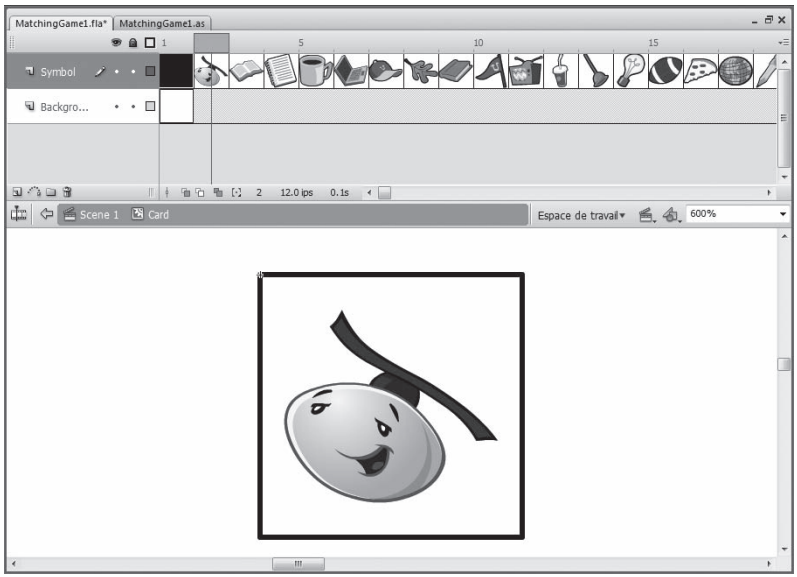


Figure 3.1
Le clip Card est un symbole avec trente-sept images. Chaque image représente une carte différente.

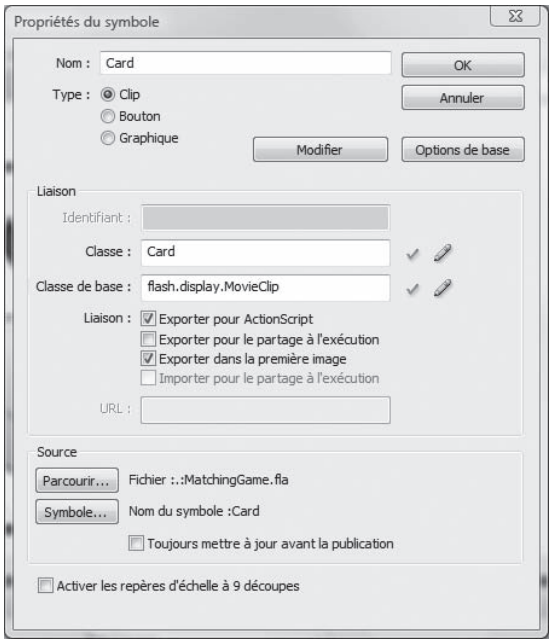


Figure 3.2
La boîte de dialogue Propriétés du symbole présente les propriétés du symbole Card.

Créer la classe ActionScript de base

Pour créer un fichier de classe ActionScript, choisissez Fichier > Nouveau, puis sélectionnez Fichier ActionScript dans la liste des types de fichiers. Vous créez ainsi un document ActionScript sans titre dans lequel vous pouvez taper votre code.

Pour commencer un fichier ActionScript 3.0, nous devons le définir comme paquetage. C'est ce que fait la première ligne de l'exemple de code suivant :

```
package {
      import flash.display.*;
```

Juste après la déclaration de paquetage, nous indiquons au moteur de lecture Flash les classes requises pour accomplir nos tâches. Dans le cas présent, nous aurons besoin d'accéder à la classe `flash.display` et à chacune de ses sous-classes immédiates. Nous pourrons ainsi créer et manipuler des clips comme les cartes.

Vient ensuite la déclaration de la classe. Le nom de la classe doit correspondre exactement à celui du fichier. Dans le cas présent, nous l'appelons `MatchingGame1`. Nous devons aussi définir l'objet que cette classe affectera. Dans le cas présent, elle affectera l'animation Flash principale, qui est un clip :

```
public class MatchingGame1 extends MovieClip {
```

Viennent ensuire les déclarations des variables qui seront utilisées dans la classe. Notre première tâche, qui consiste à créer les trente-six cartes à l'écran, est cependant si simple que nous n'aurons besoin d'aucune variable. Du moins pour l'instant.

Nous pouvons donc passer directement à la fonction d'initialisation, également appelée fonction constructeur. Cette fonction s'exécute aussitôt que la classe est créée lorsque l'animation se lit. Elle doit porter exactement le même nom que la classe et le fichier ActionScript :

```
public function MatchingGame1():void {
```

Cette fonction n'a pas besoin de retourner de valeur. Nous pouvons donc placer le mot-clé `:void` pour indiquer à Flash que rien ne sera retourné par cette fonction. Il est aussi possible de ne pas mentionner ce mot-clé, qui est par défaut implicitement déclaré par le compilateur Flash.

Dans la fonction constructeur, nous pouvons réaliser la tâche qui consiste à créer les trente-six cartes à l'écran. Nous créerons une grille de six cartes en largeur sur sixen hauteur.

Pour cela, nous utilisons deux boucles `for` imbriquées. La première fait avancer la variable `x` de 0 à 5. Le `x` représente la colonne dans notre grille de 6 par 6. Ensuite, la seconde boucle fait avancer `y` de 0 à 5, qui représente la ligne :

```
for(var x:uint=0;x<6;x++) {
      for(var y:uint=0;y<6;y++) {
```

Chacune de ces deux variables est déclarée de type `uint`, un entier non signé, juste à l'intérieur de l'instruction `for`. Chacune commence par la valeur 0, puis poursuit tant que la valeur est inférieure à 6. Elles augmentent d'une unité à chaque passage dans la boucle.

Il existe trois types de nombres : `uint`, `int` et `Number`. Le type `uint` est conçu pour tous les nombres entiers supérieurs ou égaux à 0. Le type `int` est conçu pour tous les nombres entiers qui peuvent être positifs ou négatifs. Le type `Number` peut être positif ou négatif, entier ou à virgule flottante, comme 3,5 ou –173,98. Dans les boucles `for`, nous utilisons généralement des types `uint` ou `int` car nous ne progressons que par unités complètes.

Il s'agit ainsi d'un moyen rapide d'effectuer une boucle et de créer trente-six clips Card différents. La création des clips se limite à utiliser le mot-clé `new` et `addChild`. Nous devons aussi nous assurer que, lorsque chaque clip est créé, il est arrêté sur sa première image et correctement positionné à l'écran :

```
                    var thisCard:Card = new Card();
                    thisCard.stop();
                    thisCard.x = x*52+120;
                    thisCard.y = y*52+45;
                    addChild(thisCard);
                }
            }
        }

    }
}
```

L'ajout d'un symbole dans ActionScript 3.0 ne requiert que deux commandes : `new`, qui vous permet de créer une nouvelle instance du symbole, et `addChild`, qui ajoute l'instance à la liste d'affichage pour la scène. Entre ces deux commandes, vous devez effectuer des tâches, comme définir la position `x` et `y` du nouveau symbole.

Le positionnement s'opère en fonction de la largeur et de la hauteur des cartes que nous avons créées. Dans l'animation d'exemple **MatchingGame1.fla**, les cartes font 50 pixels sur 50 avec 2 pixels d'espacement entre les cartes. En multipliant les valeurs `x` et `y` par 52, nous espaçons ainsi les cartes en intégrant un petit interstice supplémentaire. Nous ajoutons également 120 horizontalement et 45 verticalement, afin de positionner la carte à peu près au centre d'une animation Flash standard de 550 × 400.

Avant de tester ce code, nous devons lier l'animation Flash au fichier ActionScript. Le fichier ActionScript doit être enregistré sous le nom **MatchingGame1.as** et situé dans le même répertoire que l'animation **MatchingGame1.fla**.

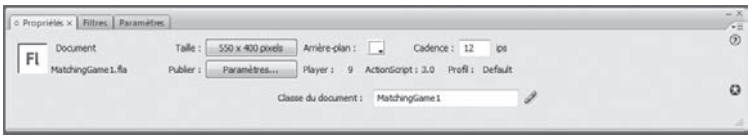


Figure 3.3
Définissez la classe du document de l'animation Flash en lui donnant le nom du fichier ActionScript contenant votre script principal.

Cette opération ne suffit cependant pas à opérer la liaison entre les deux. Vous devez également définir la propriété Classe du document dans l'inspecteur des propriétés de l'animation Flash. Sélectionnez l'onglet Propriétés de l'inspecteur des propriétés lorsque votre document d'animation **MatchingGame1.fla** est actif (voir Figure 3.3). Vous remarquerez que la classe du document est définie en bas à droite.

Vous pouvez tester votre animation lorsque l'animation Flash elle-même est active ou lorsque le document courant correspond au fichier ActionScript. Avec le fichier ActionScript, recherchez la liste déroulante Cible, située en haut à droite de la fenêtre Document. Elle indique l'animation Flash qui sera compilée et exécutée au moment du test. Si le bon fichier n'est pas sélectionné dans la liste, utilisez le menu déroulant pour modifier ce choix.

La Figure 3.4 présente l'écran après que nous avons testé l'animation. Le moyen le plus simple d'effectuer le test est d'accéder au menu et de choisir Contrôle > Tester l'animation.

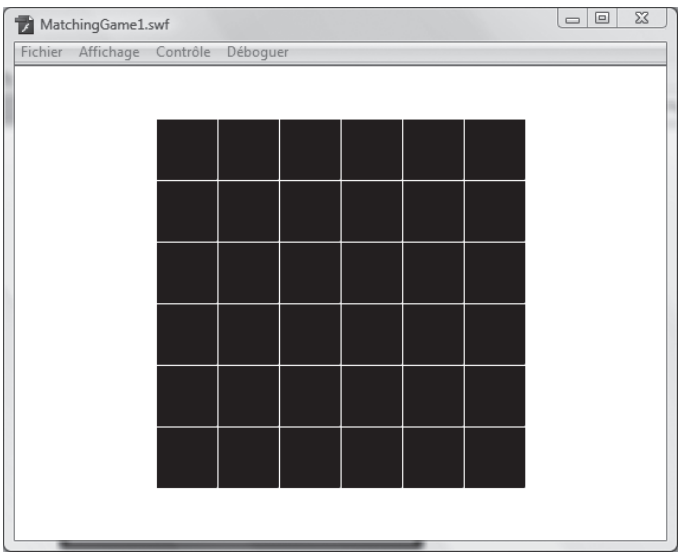


Figure 3.4
L'écran présente trente-six cartes espacées et centrées dans la scène.

Utiliser des constantes pour une meilleure programmation

Avant de poursuivre le développement de ce jeu, voyons comment améliorer ce dont nous disposons déjà. Nous allons copier l'animation existante dans le fichier **MatchingGame2.fla** et le code dans **MatchingGame2.as**. N'oubliez pas de modifier la classe de document de **MatchingGame2.fla** en choisissant MatchingGame2 et la déclaration de classe et la fonction de constructeur en la remplaçant par `MatchingGame2`.

Supposons que vous souhaitiez afficher non plus une grille de 6 × 6 cartes mais une grille plus simple de 4 × 4. Ou même une grille rectangulaire de 6 × 5. Pour cela, il vous suffit de retrouver les boucles `for` du code précédent et de les adapter pour effectuer un nombre de passages différent.

Il existe pourtant une méthode encore meilleure qui consiste à retirer ces nombres spécifiques du code. Au lieu de cela, placez-les en haut de votre code, en les étiquetant clairement, afin de pouvoir les retrouver et les modifier aisément par la suite.

Cette manière d'insérer des nombres spécifiques à l'intérieur du code, comme le 5 pour le nombre de lignes et de colonnes, est appelée "coder en dur". Cette pratique est réputée contrevenir au bon usage en programmation car elle complique les adaptations ultérieures du programme, notamment pour les programmeurs qui héritent d'un code qu'ils n'ont pas créé eux-mêmes et doivent pouvoir modifier.

Plusieurs autres valeurs codées en dur figurent dans notre programme. Dressons-en la liste :

Lignes horizontales = 6

Lignes verticales = 6

Espacement horizontal = 52

Espacement vertical = 52

Décalage écran horizontal = 120

Décalage écran vertical = 45

Au lieu de placer ces valeurs dans le code, insérons-les dans des variables constantes dans notre classe afin de pouvoir les retrouver et les modifier facilement :

```
public class MatchingGame2 extends MovieClip {
    // Constantes de jeu
    private static const boardWidth:uint = 6;
    private static const boardHeight:uint = 6;
    private static const cardHorizontalSpacing:Number = 52;
    private static const cardVerticalSpacing:Number = 52;
    private static const boardOffsetX:Number = 120;
    private static const boardOffsetY:Number = 45;
```

Vous remarquerez que j'ai choisi `private static const` lors de la définition de chacune des constantes. Le mot-clé `private` signifie que ces variables ne sont accessibles que depuis l'intérieur de cette classe. Le mot-clé `static` signifie qu'elles possèderont la même valeur dans toutes les instances de la classe. Enfin, le mot-clé `const` signifie que les valeurs ne peuvent jamais changer. Si vous utilisiez `public var` à la place, vous auriez la déclaration inverse : les variables seraient accessibles depuis l'extérieur de la classe et contiendraient des valeurs différentes pour chacune des instances. Comme il s'agit de l'unique instance de la classe et qu'il n'existe pas de script externe, il n'y a aucune différence à faire l'un ou l'autre de ces choix, mais il est plus soigneux de procéder comme nous l'avons fait.

Maintenant que nous avons des constantes, nous pouvons remplacer le code dans la fonction constructeur afin de les utiliser au lieu de nombres codés en dur :

```
public function MatchingGame2():void {
    for(var x:uint=0;x<boardWidth;x++) {
        for(var y:uint=0;y<boardHeight;y++) {
            var thisCard:Card = new Card();
            thisCard.stop();
            thisCard.x = x*cardHorizontalSpacing+boardOffsetX;
            thisCard.y = y*cardVerticalSpacing+boardOffsetY;
            addChild(thisCard);
        }
    }
}
```

Comme vous pouvez le voir, j'ai également changé le nom de la classe et la fonction en choisissant MatchingGame2. Ces exemples se trouvent dans les fichiers **MatchingGame2.fla** et **MatchingGame2.as**.

*Lorsque nous avancerons dans ce chapitre, nous modifierons les noms du fichier ActionScript et de l'animation. Si vous suivez ces étapes en créant vos propres animations de toutes pièces, n'oubliez pas non plus de modifier la classe du document dans l'inspecteur des propriétés afin que chaque animation pointe sur le bon fichier ActionScript. Par exemple, l'animation **`MatchingGame2.fla`** doit utiliser le fichier **`MatchingGame2.as`**, de sorte que sa classe de document doit être MatchingGame2.*

Ouvrez ces deux fichiers. Testez-les un à un. Ensuite, testez-les à nouveau après avoir changé certaines des constantes. Par exemple, limitez la hauteur (`boardHeight`) à cinq cartes. Faites glisser les cartes vers le bas de 20 pixels en modifiant `boardOffsetY`. Le fait que vous puissiez opérer ces modifications rapidement et sans peine est la meilleure démonstration de l'intérêt des constantes.

Mélanger et attribuer des cartes

Maintenant que nous pouvons ajouter des cartes à l'écran, il faut que nous puissions attribuer aléatoirement les images à chaque carte. S'il y a trente-six cartes à l'écran, il doit donc y avoir dix-huit paires d'images positionnées de manière aléatoire.

Au Chapitre 2, nous avons traité de l'utilisation des nombres aléatoires. Nous ne pouvons cependant pas nous contenter de choisir une image aléatoire pour chaque carte. Nous devons nous assurer qu'il y a exactement deux cartes de chaque type à l'écran. Pas une de plus, pas une de moins. Sans cela, nous n'aurons pas de véritables paires.

Ce processus est à peu près inverse de celui qui consiste à mélanger un jeu de cartes. Au lieu de mélanger les cartes et de sélectionner de nouvelles cartes à partir du haut du paquet, nous allons utiliser une liste ordonnée de cartes et sélectionner de nouvelles cartes à partir d'emplacements aléatoires du paquet.

Pour cela, nous devons créer un tableau qui liste chaque carte, puis sélectionner une carte aléatoire dans ce tableau. Le tableau fera trente-six éléments de long et contiendra deux exemplaires de chacune des dix-huit cartes. Ensuite, à mesure que nous créerons la grille de 6 × 6, nous retirerons des cartes du tableau et les placerons dans la grille. Lorsque nous aurons fini, le tableau sera vide et toutes les dix-huit paires de cartes seront prises en compte dans la grille.

Voici le code qui permet de réaliser cette opération. Une variable `i` est déclarée dans l'instruction `for`. Elle ira de zéro au nombre de cartes requis. Il s'agit simplement de la largeur de la grille multipliée par sa hauteur et divisée par deux (car chaque carte figure en deux exemplaires). Pour une grille de 6 × 6, il y aura donc trente-six cartes. Nous devons boucler dix-huit fois pour ajouter dix-huit paires de cartes :

```
// Création d'une liste de numéros de carte
var cardlist:Array = new Array();
for(var i:uint=0;i<boardWidth*boardHeight/2;i++) {
    cardlist.push(i);
    cardlist.push(i);
}
```

La commande `push` est utilisée pour placer deux fois un numéro dans le tableau. Voici à quoi ressemblera le tableau :

```
0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13,14,14,15,15,16,16,17,17
```

À présent, lorsque nous bouclons afin de créer les trente-six clips, nous récupérons un nombre aléatoire dans cette liste afin de déterminer l'image à afficher sur chaque carte :

```
for(var x:uint=0;x<boardWidth;x++) { // Horizontal
    for(var y:uint=0;y<boardHeight;y++) { // Vertical
        var c:Card = new Card(); // Copie du clip
        c.stop(); // Arrêt sur la première image
        c.x = x*cardHorizontalSpacing+boardOffsetX; // Définir la position
        c.y = y*cardVerticalSpacing+boardOffsetY;
        var r:uint = Math.floor(Math.random()*cardlist.length); // Obtenir une face aléatoire
        c.cardface = cardlist[r]; // Attribuer la face à la carte
        cardlist.splice(r,1); // Retirer la face de la list
        c.gotoAndStop(c.cardface+2);
        addChild(c); // Afficher la carte
    }
}
```

Les nouvelles lignes se trouvent au milieu du code. Pour commencer, nous utiliserons la ligne suivante pour obtenir un nombre aléatoire compris entre zéro et le nombre d'éléments qui restent dans la liste :

```
var r:uint = Math.floor(Math.random()*cardlist.length);
```

La fonction `Math.random()` retourne un nombre compris entre 0,0 et 1,0 non inclus. Nous multiplions ce résultat pour obtenir un nombre aléatoire compris entre 0,0 et 35,9999. Ensuite, nous utilisons `Math.floor()` pour arrondir ce nombre à l'entier inférieur et obtenir ainsi un nombre compris entre 0 et 35 (à tout le moins, quand il y a trente-six éléments dans le tableau `cardlist`, au début des boucles).

Ensuite, le nombre à l'emplacement correspondant dans `cardlist` est attribué à une propriété de `c` nommée `cardface`. Puis nous utilisons la commande `splice` pour retirer ce nombre du tableau afin qu'il ne soit plus utilisé de nouveau.

Le script **MatchingGame3.as** inclut en outre cette ligne pour vérifier que tout fonctionne jusque-là :

```
c.gotoAndStop(c.cardface+2);
```

Cette syntaxe amène le clip `Card` à afficher son image. Ainsi, toutes les trente-six cartes seront face dessus plutôt que face en bas. Elle prend la valeur de la propriété `cardface`, qui correspond à un nombre compris entre 0 et 17 et ajoute 2 pour obtenir un nombre compris entre 2 et 19. Cette valeur correspond aux images dans le clip `Card` : l'image 1 représente le dos des cartes et les images 2 et suivantes, leurs faces.

S'il faut généralement déclarer et définir des variables, il est également possible d'ajouter des propriétés dynamiques comme `cardface` *à un objet. Cela ne peut se faire que si l'objet est dynamique, ce que l'objet* `Card` *est par défaut car nous n'avons pas spécifié le contraire. La propriété* `cardface` *présuppose le type de la valeur qu'elle se voit attribuer (soit* `Number`*, dans le cas présent).*
Il ne s'agit pas du meilleur usage en programmation. Il serait préférable de définir une classe pour la `Card`*, avec un fichier ActionScript déclarant un paquetage, une classe, des propriétés et une fonction constructeur. Il s'agit cependant d'un considérable effort supplémentaire lorsqu'une seule petite propriété est requise ; la commodité l'emporte donc sur l'intérêt de s'en tenir aux bonnes pratiques de la programmation.*

Il n'est évidemment pas souhaitable que cette ligne de code figure dans notre jeu final, mais elle est utile à ce stade pour montrer ce que nous avons accompli. La Figure 3.5 montre à quoi pourrait ressembler l'écran une fois que nous exécutons le programme avec cette ligne de test en place.

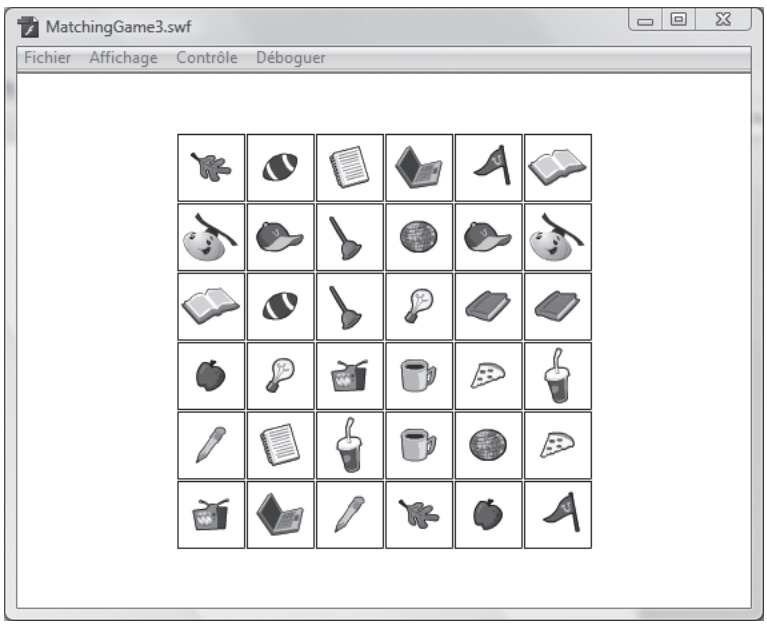


Figure 3.5
La troisième version de notre programme inclut du code qui révèle chacune des cartes. C'est utile pour obtenir une confirmation visuelle que notre code fonctionne jusque-là.

Jeu

À présent que la grille est configurée, nous devons permettre à l'utilisateur de cliquer sur des cartes pour tenter de trouver des correspondances. Nous devons également tenir le registre de l'état du jeu, ce qui implique ici de savoir si le joueur clique sur la première carte ou la seconde et si toutes les cartes ont été trouvées.

Ajouter des écouteurs clavier

La première étape consiste à amener chacune des cartes que nous créons à réagir à des clics de souris. Nous pouvons le faire en ajoutant un écouteur à chacun de ces objets. La fonction `addEventListener` s'en charge. Elle prend deux paramètres : l'événement à écouter et la fonction à appeler lorsque l'événement se produit. Voici la ligne de code :

```
c.addEventListener(MouseEvent.CLICK,clickCard);
```

Vous devez également ajouter une autre instruction `import` au début de la classe pour indiquer à Flash que vous souhaitez utiliser des événements :

```
import flash.events.*;
```

La syntaxe pour l'événement est ici `MouseEvent.CLICK`, ce qui correspond à un simple clic sur la carte. Lorsque cet événement se produit, la fonction `clickCard` est appelée. Cette fonction reste encore à créer. Nous devons la programmer avant de tester l'animation de nouveau car Flash ne peut compiler l'animation s'il manque une portion du code.

Voici un début simple pour la fonction `clickCard` :

```
public function clickCard(event:MouseEvent) {
    var thisCard:Card = (event.currentTarget as Card); // Quelle carte ?
    trace(thisCard.cardface);
}
```

L'utilisation d'un appel à l'instruction `trace` peut être un excellent moyen de vérifier votre code afin d'avancer par petites étapes et d'éviter les maux de tête. Par exemple, si vous ajoutez vingt-sept lignes de code à la fois et qu'ensuite le programme ne fonctionne pas comme prévu, vous devez localiser le problème dans vingt-sept nouvelles lignes de code. Si vous n'ajoutez que cinq nouvelles lignes de code et utilisez une instruction `trace` pour afficher les valeurs des variables clés, vous pouvez résoudre tous les problèmes avec ces cinq lignes de code avant de passer à la suite.

À chaque fois qu'une fonction réagit à un événement, elle doit prendre au moins un paramètre, à savoir l'événement lui-même. Dans le cas présent, il s'agit d'une valeur de type `MouseEvent` que nous attribuerons à la variable `event`.

Vous devez accepter le paramètre d'événement ou une fonction d'écouteur événementiel, que vous vous souciez de sa valeur ou non. Par exemple, si vous créez un unique bouton et savez que la fonction ne s'exécutera que lorsque ce bouton sera enfoncé, vous devez malgré tout accepter l'événement comme paramètre, même si vous ne l'utilisez pour rien ensuite.

Dans le cas présent, le paramètre `event` est essentiel parce que nous devons savoir sur laquelle des trente-six cartes le joueur a cliqué. La valeur du paramètre `event` est en fait un objet avec toutes sortes de propriétés, mais la seule chose qui nous intéresse est de connaître l'objet `Card` sur lequel l'utilisateur a cliqué. Il s'agit de la cible ou, plus précisément, de la propriété `currentTarget` de l'événement.

`currentTarget` est cependant un objet vague pour le moteur ActionScript à ce stade. Bien sûr, il s'agit d'un objet `Card`. Il s'agit cependant aussi d'un clip, autrement dit d'un objet d'affichage également. Nous souhaitons obtenir sa valeur en tant qu'objet `Card` et définissons donc une variable en tant que `Card`, puis utilisons un `Card` pour spécifier que nous souhaitons que la valeur de `event.currentTarget` soit retournée sous forme de `Card`.

Maintenant que nous avons un objet `Card` dans la variable `thisCard`, nous pouvons accéder à sa propriété `cardface`. Nous utiliserons `trace` pour la placer dans la fenêtre Sortie et exécuter un test rapide de **MatchingGame4.fla** pour nous assurer que l'animation fonctionne.

Configurer la logique du jeu

Lorsqu'un joueur clique sur une carte, nous devons déterminer les étapes à suivre en fonction de son choix et de l'état du jeu. Il existe trois états principaux à gérer :

- **État 1.** Aucune carte n'a été choisie, le joueur sélectionne la première carte d'une correspondance possible.
- **État 2.** Une carte a été choisie, le joueur sélectionne une seconde carte. Une comparaison doit être opérée et une action, entreprise en fonction du fait qu'il s'agit ou non d'une correspondance.
- **État 3.** Deux cartes ont été choisies, mais aucune correspondance n'a été trouvée. Conserver ces cartes face visible jusqu'à ce qu'une nouvelle carte soit choisie, puis les retourner toutes les deux et révéler la nouvelle carte.

Les Figures 3.6 à 3.8 présentent les trois états du jeu.

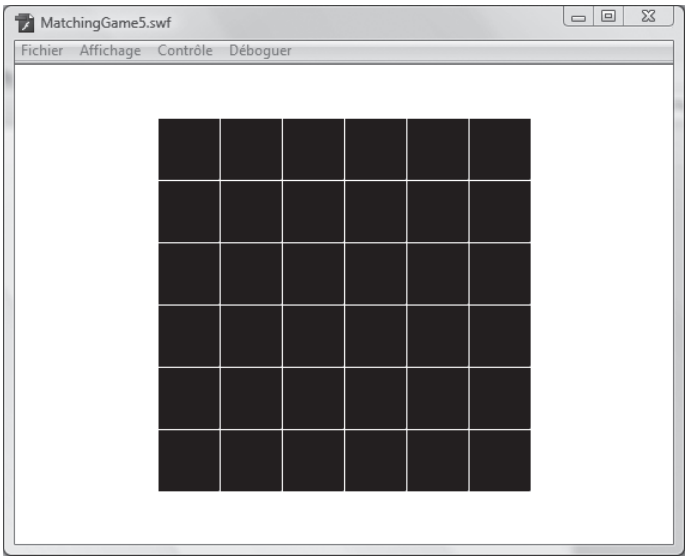


Figure 3.6
État 1, où l'utilisateur est sur le point de choisir sa première carte.

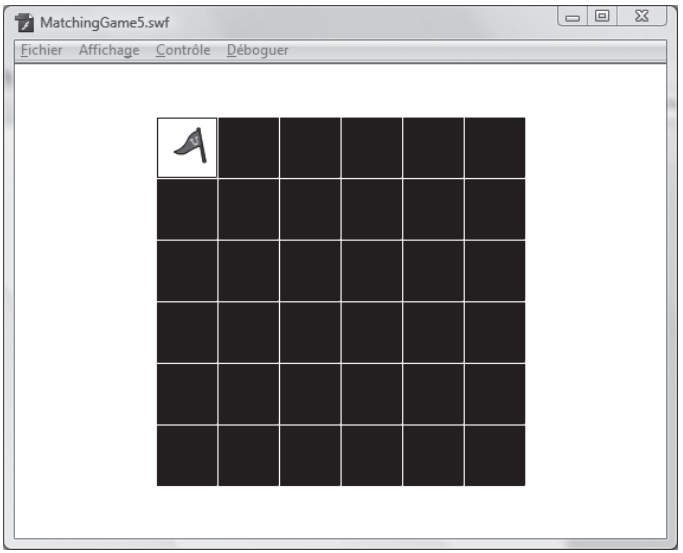


Figure 3.7
État 2, où l'utilisateur est sur le point de choisir sa seconde carte.

D'autres considérations entrent ensuite en jeu. Que se passe-t-il si le joueur clique sur une carte puis clique sur la même carte de nouveau ? Nous en déduisons que le joueur souhaite probablement revenir sur son premier choix : nous devons retourner cette carte et revenir au premier état.

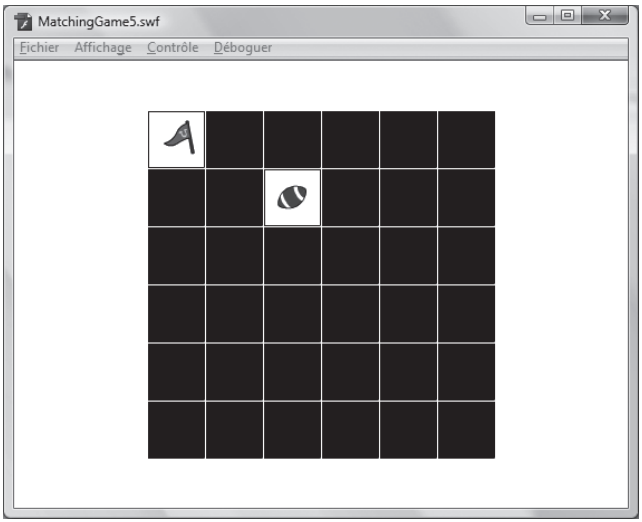


Figure 3.8
État 3, où une paire de cartes a été sélectionnée mais aucune correspondance, trouvée. L'utilisateur doit maintenant choisir une autre carte pour tenter de trouver une autre paire.

Nous devrons inévitablement tenir le registre des cartes qui ont été choisies lorsque le joueur tombe sur une correspondance. Voilà donc l'occasion de créer nos premières variables de classe. Nous les appellerons `firstCard` et `secondCard`. Elles seront toutes deux de type `Card` :

```
private var firstCard:Card;
private var secondCard:Card;
```

Comme nous n'avons défini aucune valeur pour ces variables, elles commenceront toutes deux avec la valeur d'objet par défaut `null`. En fait, nous utiliserons les valeurs `null` de ces deux variables pour déterminer l'état.

Tous les types de variables ne peuvent pas être positionnés à `null`. Par exemple, une variable `int` sera positionnée à zéro au moment de sa création, à moins que vous ne spécifiiez une autre valeur. Vous ne pouvez pas la positionner à `null` même si vous le souhaitez.

Si `firstCard` et `secondCard` valent toutes deux `null`, nous devons nous trouver au premier état. Le joueur est sur le point de choisir sa première carte.

Si `firstCard` ne vaut pas `null` et `secondCard` vaut `null`, nous nous trouvons au second état. Le joueur choisira bientôt la carte qu'il espère correspondre à la première.

Si `firstCard` et `secondCard` ne valent `null` ni l'une ni l'autre, nous nous trouvons au troisième état. Nous utiliserons les valeurs `firstCard` et `secondCard` pour savoir quelles cartes retourner lorsque l'utilisateur choisira la `firstCard` suivante.

Examinons le code :

```
public function clickCard(event:MouseEvent) {
    var thisCard:Card = (event.target as Card); // Quelle carte ?

    if (firstCard == null) { // Première carte dans une paire
        firstCard = thisCard; // La mémoriser
        firstCard.gotoAndStop(thisCard.cardface+2); // La retourner
```

Jusque-là, vous pouvez voir ce qui se passe lorsque le joueur clique sur la première carte. Vous remarquerez que la commande `gotoAndStop` est analogue à celle que nous avons utilisée pour tester le mélange des cartes précédemment dans ce chapitre. Elle doit ajouter 2 au numéro de l'image afin que les valeurs de carte comprises entre 0 et 17 correspondent aux numéros d'image compris entre 2 et 19 qui contiennent les dix-huit faces de carte.

À présent que la valeur de `firstCard` est définie, nous pouvons attendre le second clic. Cette étape est gérée par les deux parties suivantes de l'instruction `if`. Cette partie gère le cas où le joueur clique sur la première carte de nouveau et retourne cette carte en redonnant la valeur `null` à `firstCard` :

```
    } else if (firstCard == thisCard) { // Première carte cliquée de nouveau
        firstCard.gotoAndStop(1); // Retourner
        firstCard = null;
```

Si le joueur clique sur une autre carte afin de découvrir la seconde, une comparaison doit être effectuée entre les deux. Nous comparons non pas les cartes elles-mêmes mais leur propriété `cardface`. Si les faces sont les mêmes, une correspondance a été trouvée :

```
    } else if (secondCard == null) { // Seconde carte dans une paire
        secondCard = thisCard; // La mémoriser
        secondCard.gotoAndStop(thisCard.cardface+2); // La retourner

        // Comparer les deux cartes
        if (firstCard.cardface == secondCard.cardface) {
```

Si une correspondance a été trouvée, nous souhaitons supprimer les cartes et réinitialiser les variables `firstCard` et `secondCard` : il faut pour cela utiliser la commande `removeChild`, qui est l'inverse de `addChild`. Elle retire l'objet de la liste d'affichage et le supprime de l'affichage. Dans le cas présent, les objets se trouvent cependant toujours stockés dans des variables, aussi devons-nous les positionner à `null` pour qu'ils soient supprimés par le lecteur Flash :

```
            // Supprimer une correspondance
            removeChild(firstCard);
            removeChild(secondCard);
            // Réinitialiser sélection
            firstCard = null;
            secondCard = null;
        }
```

Le cas suivant correspond à ce qui se passe si le joueur a sélectionné une `firstCard` puis sélectionné une seconde carte qui ne lui correspond pas et clique à présent sur une nouvelle carte. Cette opération doit retourner les deux premières cartes en les ramenant à leur position face cachée, ce qui correspond à l'image 1 du clip `Card`.

Immédiatement après, elle doit attribuer la nouvelle carte à `firstCard` et afficher son image :

```
        } else { // Recherche d'une autre paire
            // Réinitialisation de la paire précédente
            firstCard.gotoAndStop(1);
            secondCard.gotoAndStop(1);
            secondCard = null;
            // Sélection de la première carte de la paire suivante
            firstCard = thisCard;
            firstCard.gotoAndStop(thisCard.cardface+2);
        }
    }
```

Voilà tout, en fait, pour le jeu de base. Vous pouvez tester **MatchingGame5.fla** et **MatchingGame5.as** pour y jouer. Vous pouvez sélectionner des paires de cartes et voir comment les cartes correspondantes sont retirées de la grille.

Ce jeu peut être considéré comme complet. Vous pourriez aisément placer une image à l'arrière-plan des cartes dans le scénario de l'animation principale et faire de la découverte de l'image complète la récompense pour le gain du jeu. Comme simple gadget dans un site Web, cela suffira parfaitement. Il est pourtant possible d'aller bien plus loin et d'ajouter d'autres fonctionnalités.

Vérifier si la partie est finie

Vous souhaiterez sans nul doute vérifier si la partie est finie pour récompenser les joueurs en affichant un écran indiquant qu'ils ont terminé le jeu. L'état de fin de partie est atteint lorsque toutes les cartes ont été supprimées.

Dans les exemples de ce chapitre, nous conduisons simplement le joueur vers un écran qui affiche les mots Game Over. Vous pourriez cependant afficher une animation et conduire à une nouvelle page Web. Nous nous en tiendrons néanmoins ici exclusivement à ce qui concerne la programmation des jeux.

Il existe plusieurs approches pour ce travail. Par exemple, vous pouvez avoir une nouvelle variable dans laquelle vous tenez le registre du nombre de paires trouvées. À chaque fois que vous trouvez une paire, ajoutez une unité à cette valeur, puis vérifiez-la pour voir si elle est égale au nombre total de paires.

Une autre méthode consiste à vérifier la propriété `numChildren` de l'objet `MatchingGame`. Lorsque vous lui ajoutez trente-six cartes, `numChildren` vaut 36. À mesure que des paires sont supprimées, `numChildren` s'avance vers zéro. Lorsque sa valeur atteint zéro, la partie est terminée.

Le problème avec cette méthode tient à ce que si vous placez d'autres éléments dans la scène, comme un arrière-plan ou une barre de titre, ils sont également comptés dans `numChildren`.

Pour le présent exemple, la première méthode semble préférable. Au lieu de compter le nombre de cartes supprimées, comptons le nombre de cartes affichées. Créons une nouvelle variable de classe nommée `cardsLeft` :

```
private var cardsLeft:uint;
```

Ensuite, positionnons-la à zéro juste avant les boucles `for` qui créent les cartes et ajoutons une unité à cette variable pour chaque carte créée :

```
cardsLeft = 0;
for(var x:uint=0;x<boardWidth;x++) { // Horizontale
    for(var y:uint=0;y<boardHeight;y++) { // Verticale
        var c:Card = new Card(); // Copier le clip
        c.stop(); // Arrêt sur la première image
        c.x = x*cardHorizontalSpacing+boardOffsetX; // Définir la position
        c.y = y*cardVerticalSpacing+boardOffsetY;
        var r:uint = Math.floor(Math.random()*cardlist.length); // Obtenir une face aléatoire
        c.cardface = cardlist[r]; // Attribuer face à la carte
        cardlist.splice(r,1); // Supprimer face de la liste
        c.addEventListener(MouseEvent.CLICK,clickCard); // Écouter les clics
        addChild(c); // Afficher la carte
        cardsLeft++;
    }
}
```

Nous devons ensuite ajouter du nouveau code lorsque l'utilisateur trouve une correspondance et que les cartes sont supprimées de l'écran. Il vient se placer dans la fonction `clickCard`.

```
cardsLeft -= 2;
if (cardsLeft == 0) {
    gotoAndStop("gameover");
}
```

Vous pouvez utiliser `++` pour ajouter une unité à une variable et `--` pour la soustraire. Par exemple, `cardsLeft++` équivaut à `cardsLeft = cardsLeft + 1`.
Vous pouvez également utiliser `+=` pour ajouter un nombre à une variable et `-=` pour le soustraire. Par exemple, `cardsLeft -= 2` équivaut à `cardsLeft = cardsLeft - 2`.

Voilà tout ce dont nous avions besoin pour la programmation du code. À présent, le jeu mémorise le nombre de cartes à l'écran à l'aide de la variable `cardsLeft` et intervient lorsque ce nombre atteint zéro.

Figure 3.9
L'écran de fin de partie le plus simple du monde.

L'action effectuée consiste à sauter à une nouvelle image comme celle présentée à la Figure 3.9. Si vous examinez l'animation **MatchingGame6.fla**, vous verrez que j'ai ajouté une seconde image. J'ai également ajouté des commandes `stop();` à la première image. Elles arrêtent l'animation sur la première image afin que l'utilisateur puisse jouer sa partie au lieu de poursuivre à l'image suivante. La seconde image est étiquetée `gameover` et sera utilisée lorsque la propriété `cardsLeft` vaudra zéro.

À ce stade, nous souhaitons supprimer tous les éléments de jeu créés par le code. Comme le jeu ne crée que trente-six cartes et que toutes les trente-six cartes sont supprimées lorsque le joueur trouve toutes les correspondances, il ne reste néanmoins aucun élément supplémentaire à supprimer à l'écran. Nous pouvons sauter à l'image `gameover` sans craindre qu'il ne reste d'élément à l'écran.

L'écran `gameover` affiche les mots Game Over dans l'animation d'exemple. Vous pouvez y ajouter des images ou des animations sophistiquées si vous le souhaitez. Dans la suite de ce chapitre, nous verrons comment ajouter un bouton Play Again (Rejouer) à cette image.

Encapsuler le jeu

Nous disposons maintenant d'un jeu qui s'exécute sous la forme d'une animation Flash complète. L'animation est appelée **MatchingGameX.fla** et la classe ActionScript, **MatchingGameX.as**. Lorsque l'animation s'exécute, le jeu s'initialise et la partie démarre. L'animation équivaut au jeu et le jeu, à l'animation.

Ce dispositif fonctionne bien dans des situations simples. En pratique, il est cependant souhaitable d'avoir des écrans d'introduction, des écrans de fin de partie, des écrans de chargement, etc. Vous pourriez même souhaiter avoir différents écrans avec différentes versions du jeu ou des jeux complètement différents.

Flash excelle à l'encapsulation. Les animations Flash sont des clips et vous pouvez insérer des clips à l'intérieur d'autres clips. Un jeu peut ainsi être une animation ou bien un clip à l'intérieur d'une animation.

Pourquoi procéder de cette manière ? Pour une raison simple : il est aisé d'ajouter d'autres écrans à votre jeu. Il est possible de faire de l'image 1 un écran d'introduction, de l'image 2 un jeu et de l'image 3 l'écran de fin de partie.

L'image 2 contiendra en fait un clip appelé `MatchingGameObject7` qui utilise la classe **MatchingGameObject7.as**.

La Figure 3.10 présente un diagramme des trois images que nous prévoyons d'inclure dans notre animation mise à jour et du contenu de chacune d'entre elles.

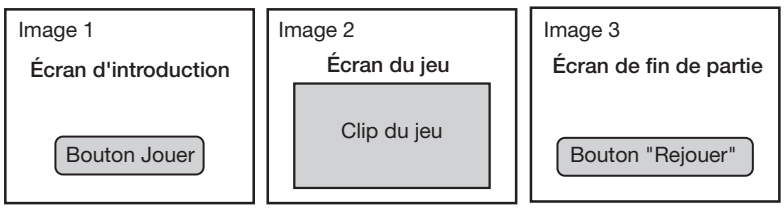


Figure 3.10
La seconde image de l'animation contient un clip qui correspond au jeu lui-même. Les autres images peuvent contenir des écrans divers.

Créer le clip du jeu

MatchingGameObject7.fla contient trois images. Passons directement à la deuxième. Elle contient un unique clip. Vous ne l'aurez peut-être pas remarqué au premier abord parce qu'il s'agit d'un clip entièrement vide qui apparaît donc sous la forme d'un petit cercle dans le coin supérieur gauche de l'écran.

Dans la bibliothèque, ce clip est nommé `MatchingGameObject7` et, comme le montre la Figure 3.11, la classe **MatchingGameObject7** lui est associée.

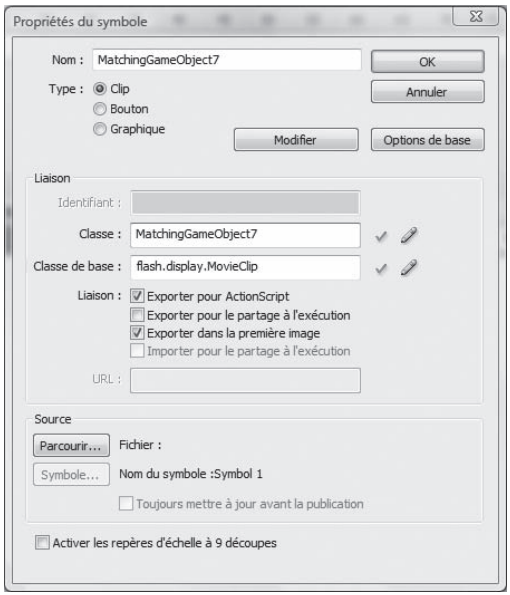


Figure 3.11
Ce clip utilisera le fichier MatchingGameObject7.as comme classe.

En somme, ce clip remplace le jeu entier et le scénario de l'animation principal correspond maintenant à un clip plus grand qui l'englobe.

Lorsque l'animation atteint l'image 2, le clip `MatchingGameObject7` prend vie, exécute la fonction constructeur de classe dans sa classe **MatchingGameObject7.as** et la lecture du jeu se lance dans ce clip.

Lorsque l'animation passe à l'image 3, le jeu entier disparaît car le clip n'existe qu'à l'image 2.

Ce dispositif permet de placer des images avant et après le jeu (et donc d'isoler le code du jeu afin de ne s'occuper que du jeu lui-même).

Ajouter un écran d'introduction

La plupart des jeux proposent un écran d'introduction. On préfère généralement que le joueur ne soit pas immédiatement plongé dans le jeu mais commence par une introduction ou puisse lire des instructions.

L'écran d'introduction contiendra du code dans l'image 1 du scénario principal. Ce code doit d'abord arrêter l'animation afin qu'elle ne poursuive pas au-delà de l'image 1. Ensuite, il doit configurer un bouton afin de permettre aux utilisateurs de démarrer le jeu.

Si vous souhaitez conserver tout le code à l'écart du scénario principal, vous pouvez configurer un nouveau fichier de classe AS comme classe de document de l'animation complète. Ce code s'exécutera à l'image 1 et peut réaliser les mêmes tâches que dans le scénario principal. Il est cependant tellement simple d'ajouter ce petit bout de code au scénario principal et d'éviter ainsi de créer plus de fichiers que nécessaire…

Le script de l'image doit d'abord attribuer un écouteur à un bouton que nous allons créer dans la première image. Nous nommerons ce bouton `playButton`.

L'écouteur événementiel appellera la fonction `startGame`, qui émet simplement une commande `gotoAndStop` à destination du scénario principal afin de lui ordonner d'atteindre l'image appelée `playgame`, qui correspond à l'image 2.

Nous placerons également une commande `stop` dans l'image afin que, lorsque l'animation s'exécute, elle s'arrête à l'image 1 et attende que l'utilisateur clique sur le bouton :

```
playButton.addEventListener(MouseEvent.CLICK,startGame);

function startGame(event:MouseEvent) {
    gotoAndStop("playgame");
}

stop();
```

Dans la seconde image se trouvera le clip `MatchingGameObject7`. Ensuite, nous devrons renommer le fichier AS de classe de document **MatchingGameObject7.as** afin qu'il soit utilisé par ce clip et non par l'animation principale.

Pour créer un clip vide, accédez à la bibliothèque et choisissez Nouveau symbole dans son menu. Nommez le symbole, choisissez le type Clip et définissez ses propriétés. Ensuite, faites glisser le clip de la bibliothèque vers la scène. Placez-le dans le coin supérieur gauche afin que sa position 0, 0 soit la même que la position 0, 0 de la scène.

Nous devons opérer un changement dans le code. Une référence au scénario principal est utilisée lorsque la partie se termine. La commande `gotoAndStop` ne fonctionne plus correctement car le jeu se trouve dans le clip alors que l'image `gameover` se trouve dans le scénario principal. Il convient de faire la modification suivante :

```
MovieClip(root).gotoAndStop("gameover");
```

Pourquoi n'est-il pas possible d'écire simplement `root.gotoAndStop("gameover")` *? Après tout,* `root` *correspond bien au scénario principal et au parent du clip. Le compilateur ActionScript ne le permettra cependant pas. La commande* `gotoAndStop` *ne peut être émise qu'à destination de clips et, techniquement,* `root` *peut être autre chose qu'un clip, comme un clip à une image appelé sprite. Afin de rassurer le compilateur sur le fait que* `root` *est un clip, nous le tapons donc explicitement en utilisant la fonction* `MovieClip()`.

L'image `gameover` de l'animation est pour l'instant la même que dans **MatchingGame6.fla**. Il s'agit simplement d'une image contenant les mots Game Over.

L'animation **MatchingGame7.fla** est un peu différente des six précédentes versions car aucune classe de document ne lui est associée. En fait, il n'existe même pas de fichier **MatchingGame7.as**. Observez de près la structure de cette animation ainsi que la Figure 3.10 pour comprendre comment le jeu tient dans l'animation principale plus importante.

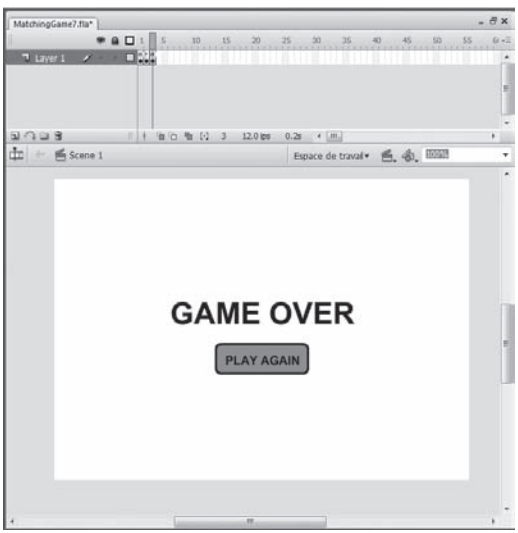
Ajouter un bouton pour rejouer

Dans la dernière image, nous souhaitons ajouter un autre bouton qui permette aux joueurs de rejouer.

Il suffit de dupliquer le bouton Play d'origine provenant de l'image 1. Ne vous contentez pas d'une opération de copier-coller. Au lieu de cela, créez un duplicata du bouton dans la bibliothèque. Ensuite, changez le texte du bouton en remplaçant Play par Play Again.

Votre image `gameover` devrait maintenant ressembler à celle de la Figure 3.12.

Figure 3.12
L'écran gameover contient maintenant un bouton Play Again.



Une fois que vous avez ajouté ce bouton à la troisième image, nommez-le `playAgainButton` à l'aide de l'inspecteur des propriétés afin de pouvoir lui attribuer un écouteur. Le script de l'image devrait ressembler à ceci :

```
playAgainButton.addEventListener(MouseEvent.CLICK,playAgain);

function playAgain(event:MouseEvent) {
    gotoAndStop("playgame");
}
```

Testez **MatchingGame7.fla** afin de voir ces boutons en action. Vous disposez maintenant d'une structure de jeu très flexible où vous pouvez substituer du contenu dans les pages `intro` et `gameover` et redémarrer le jeu sans craindre que traînent des éléments d'écran ou des valeurs de variable restants. C'était un problème avec ActionScript 1 et 2, mais ce n'est pas le cas avec ce type de framework dans ActionScript 3.0.

Ajouter un score et un chronomètre

Le but de ce chapitre est de développer une structure de jeu complète autour d'un jeu de Memory. Parmi les éléments qui se rencontrent fréquemment dans la plupart des jeux figurent le score et le chronomètre. Si le principe du Memory ne les requiert pas vraiment, nous les ajouterons tout de même afin de disposer des fonctionnalités les plus complètes possibles.

Ajouter le score

Le problème consiste à décider de la manière de comptabiliser le score pour un jeu de ce type. Il n'existe pas de solution évidente. Il devrait cependant y avoir une récompense pour l'obtention d'une correspondance et éventuellement une sanction en cas d'échec. Comme il arrive presque toujours que le joueur rate quelques coups de plus qu'il n'en réussit, il faut que les correspondances rapportent plus de points que n'en coûtent les tentatives manquées. Pour commencer, on peut envisager d'attribuer 100 points pour une correspondance et -5 en cas d'échec.

Au lieu de coder ces montants en dur dans le jeu, ajoutons-les à la liste de constantes au début de la classe :

```
private static const pointsForMatch:int = 100;
private static const pointsForMiss:int = -5;
```

Il nous faut maintenant un champ texte pour afficher le score. Comme vous l'avez vu au Chapitre 2, il est assez facile de créer un champ texte. Nous devons d'abord déclarer un nouvel objet `TextField` dans la liste des variables de classe :

```
private var gameScoreField:TextField;
```

Ensuite, nous devons créer ce champ texte et l'ajouter comme enfant :

```
gameScoreField = new TextField();
addChild(gameScoreField);
```

Nous pourrions aussi le formater et l'habiller quelque peu, comme nous l'avons fait au Chapitre 2, mais nous laisserons de côté cette tâche pour l'instant.

Le score lui-même sera une simple variable entière nommée `gameScore`. Nous la déclarerons au début de la classe :

```
private var gameScore:int;
```

Ensuite, nous la positionnerons à zéro dans la fonction constructeur :

```
gameScore = 0;
```

En outre, il serait intéressant d'afficher immédiatement le score dans le champ texte :

```
gameScoreField.text = "Score: "+String(gameScore);
```

Nous réalisons à ce stade qu'il y a au moins plusieurs endroits dans le code où nous allons définir le texte de `gameScoreField`. Le premier est la fonction constructeur. Le second intervient après que le score a changé pendant que la partie est en cours. Au lieu de copier et de coller la ligne de code précédente à deux endroits, plaçons-la dans une fonction à part. Nous pourrons alors appeler la même fonction depuis chacun des endroits du code où nous devons mettre à jour le score :

```
public function showGameScore() {
    gameScoreField.text = "Score: "+String(gameScore);
}
```

Nous devons changer le code à deux endroits. Le premier vient juste après qu'une correspondance eut été trouvée et juste avant que nous vérifiions si la partie est terminée :

```
gameScore += pointsForMatch;
```

Ensuite, nous ajoutons une clause `else` à l'instruction `if` qui vérifie s'il y a correspondance et soustrait les points en cas d'échec :

```
gameScore += pointsForMiss;
```

Voici la section de code entière afin de repérer où s'insèrent ces deux lignes :

```
// Comparer les deux cartes
if (firstCard.cardface == secondCard.cardface) {
    // Retirer une correspondance
    removeChild(firstCard);
    removeChild(secondCard);
    // Réinitialiser la sélection
    firstCard = null;
```

```
            secondCard = null;
            // Ajouter des points
            gameScore += pointsForMatch;
            showGameScore();
            // Vérifier si la partie est terminée
            cardsLeft -= 2; // 2 cartes en moins
            if (cardsLeft == 0) {
                MovieClip(root).gotoAndStop("gameover");
            }
        } else {
            gameScore += pointsForMiss;
            showGameScore();
        }
```

Vous remarquerez que nous ajoutons des points à l'aide de l'opérateur `+=`, même en cas de tentative manquée. La variable `pointsForMiss` vaut en effet –5 et l'ajout de –5 équivaut à soustraire 5 points.

Nous plaçons également l'appel à la fonction `showGameScore()` après chaque changement du score. Nous nous assurons ainsi que le joueur verra un score à jour, comme le montre la Figure 3.13.

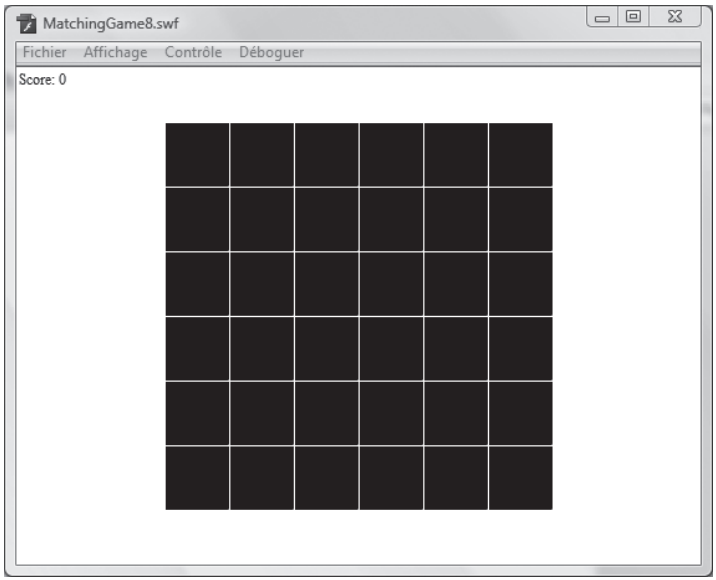


Figure 3.13
Le score apparaît maintenant dans le coin supérieur gauche avec la police et le style par défaut.

MatchingGame8.fla et **MatchingGame8.as** incluent ce code de score. Examinez-les pour découvrir cet exemple en action.

*En passant de **MatchingGame8.fla** à **MatchingGame9.fla**, vous devez faire plus que changer les noms de fichiers. Dans l'animation, vous devez changer à la fois le nom et la classe du clip* `MatchingGameObject7` *en choisissant* `MatchingGameObject8`*. Ne commettez pas l'erreur de ne changer que le nom du clip et de laisser la classe pointer vers* `MatchingGameObject7`*.*

*Ensuite, vous devez évidemment changer le nom du fichier ActionScript pour le remplacer par **MatchingGame8.as** et changer le nom de la classe et de la fonction constructeur également.*

Cela vaut également pour les prochaines versions du jeu de Memory dans la suite du chapitre.

Ajouter un chronomètre

L'ajout d'un chronomètre est un peu plus compliqué que le score. Tout d'abord, le chronomètre doit constamment être mis à jour, à la différence du score, qui ne se modifie que lorsque l'utilisateur tente de trouver une paire.

Pour inclure un chronomètre, nous devons utiliser la fonction `getTimer()`. Elle retourne le temps écoulé en millisecondes depuis que l'animation Flash a démarré. Il s'agit d'une fonction spéciale qui requiert l'importation d'une classe Flash au début de notre programme :

```
import flash.utils.getTimer;
```

La fonction `getTimer` *mesure le nombre de millisecondes écoulé depuis le début de l'animation Flash. En général, cette mesure brute n'est cependant pas utile telle quelle car le joueur ne commence pas son jeu dès l'instant où l'animation apparaît à l'écran.* `getTimer` *est plutôt utile lorsque l'on récupère deux mesures et soustrait la première à la deuxième. C'est ce que nous allons faire ici : obtenir le temps où l'utilisateur a cliqué sur Play et soustraire cette valeur au temps courant pour obtenir la durée depuis laquelle la partie a commencé.*

Il nous faut à présent de nouvelles variables. Nous avons besoin d'enregistrer le temps depuis lequel la partie a commencé. Ensuite, nous pourrons simplement soustraire cette mesure au temps courant pour obtenir la durée écoulée de la partie. Nous utiliserons aussi une variable pour stocker le temps de partie :

```
private var gameStartTime:uint;
private var gameTime:uint;
```

Nous devons aussi définir un nouveau champ texte pour afficher le temps à l'écran :

```
private var gameTimeField:TextField;
```

Dans la fonction constructeur, nous ajoutons un nouveau champ texte pour afficher le temps écoulé. Nous nous déplaçons en outre à droite de l'écran pour que ce champ texte ne se trouve pas par-dessus celui qui affiche le score :

```
gameTimeField = new TextField();
gameTimeField.x = 450;
addChild(gameTimeField);
```

Avant que la fonction constructeur n'ait terminé, nous souhaitons définir la variable `gameStartTime`. Nous pouvons aussi positionner `gameTime` à zéro :

```
gameStartTime = getTimer();
gameTime = 0;
```

Nous devons maintenant trouver un moyen de mettre à jour le temps de jeu. Ce temps change constamment : nous ne souhaitons donc pas attendre que l'utilisateur agisse pour l'afficher.

L'une des approches possibles consiste à créer un objet `Timer`, comme au Chapitre 2. Il n'est cependant pas essentiel que le chronomètre soit mis à jour à intervalles réguliers ; il faut simplement qu'il le soit suffisamment souvent pour que les joueurs aient un aperçu précis du temps de jeu écoulé.

Au lieu d'utiliser un `Timer`, nous pouvons amener l'événement `ENTER_FRAME` à déclarer une fonction qui met à jour le chronomètre. Dans une animation Flash par défaut, cela se produit douze fois par seconde, ce qui suffit amplement :

```
addEventListener(Event.ENTER_FRAME,showTime);
```

Il ne reste plus qu'à créer la fonction `showTime`. Elle calculera le temps courant en fonction de la valeur courante de `getTimer()` et de la valeur de `gameStartTime`. Ensuite, elle placera le résultat dans le champ texte afin de l'afficher :

```
public function showTime(event:Event) {
    gameTime = getTimer()-gameStartTime;
    gameTimeField.text = "Time: "+gameTime;
}
```

La Figure 3.14 présente l'écran avec le score et le temps de jeu. Le format de temps utilise cependant un signe deux-points et deux chiffres pour les secondes. C'est la prochaine étape que nous allons étudier.

Afficher le temps

La fonction `showTime` affiche le nombre de millisecondes depuis le début de la partie. Les joueurs ne se préoccupent généralement pas des millisecondes ; ils souhaitent pouvoir observer un affichage de chronomètre ou d'horloge normal où sont indiquées les minutes et les secondes, comme sur les montres digitales.

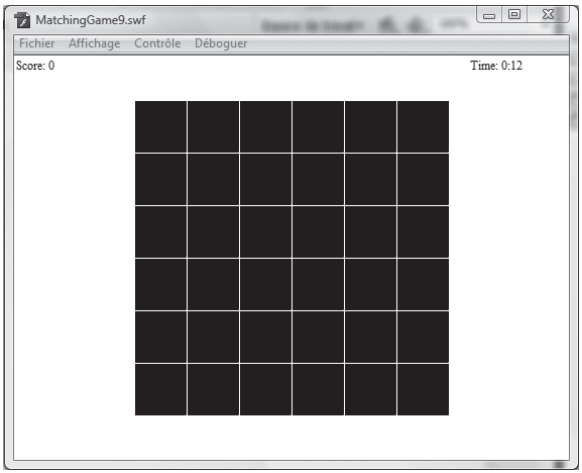


Figure 3.14
Le temps écoulé est maintenant affiché en haut à droite.

Décomposons cette tâche en une autre fonction. Au lieu d'inclure directement dans le champ texte la mesure brute `gameTime` comme dans l'exemple de code précédent, nous pouvons appeler une fonction permettant de retourner une sortie plus élégante :

```
gameTimeField.text = "Time: "+clockTime(gameTime);
```

Quand l'ancien code affiche ceci :

```
Time: 123726
```

Le nouveau code affiche cette fois :

```
Time: 2:03
```

La fonction `clockTime` récupère le temps en millisecondes brutes et le convertit en minutes et en secondes entières. En outre, elle formate cette mesure en utilisant un signe deux-points (:) et s'assure qu'un zéro est correctement placé lorsque le nombre de secondes est inférieur à dix.

La fonction commence en divisant simplement le nombre de millisecondes par 1 000 afin d'obtenir le nombre de secondes. Elle divise ensuite ce résultat par 60 pour obtenir le nombre de minutes.

Ensuite, elle doit soustraire les minutes des secondes. Par exemple, 123 secondes remplissent 2 minutes. En soustrayant 2 × 60 à 123, on obtient 3 secondes restantes. 123 correspond donc à 2 minutes et 3 secondes :

```
public function clockTime(ms:int) {
    var seconds:int = Math.floor(ms/1000);
    var minutes:int = Math.floor(seconds/60);
     seconds -= minutes*60;
```

À présent que nous avons le nombre de minutes et de secondes, nous devons insérer un signe deux-points entre les deux et veiller à ce que les secondes soient toujours exprimées à l'aide de deux chiffres.

Je me sers d'une astuce pour cela. La fonction `substr` permet de récupérer un nombre défini de caractères d'une chaîne. Le nombre de secondes sera compris entre 0 et 59. Ajoutez-y 100 et vous obtenez un nombre compris entre 100 et 159. Récupérez le second et le troisième caractère de ce nombre sous forme de chaîne et vous obtenez un nombre de secondes à deux chiffres compris entre 00 et 59. Voici comment se traduit ceci en ActionScript :

```
var timeString:String = minutes+":"+String(seconds+100).substr(1,2);
```

Ensuite, nous retournons simplement la valeur :

```
    return timeString;
}
```

Le temps écoulé s'affiche maintenant en haut de l'écran dans un format numérique familier au lieu de représenter un nombre rébarbatif de millisecondes.

Afficher le score et le temps une fois la partie terminée

Avant d'en terminer avec **MatchingGame9.fla**, récupérons les nouveaux affichages du score et du temps et transportons-les dans l'écran `gameover`.

Cette opération est un peu périlleuse car l'image `gameover` se situe dans le scénario principal, en dehors du clip du jeu. Pour que le scénario principal sache même ce que sont le score et le temps, ces données doivent être transmises du niveau du jeu vers le niveau racine.

Juste avant d'appeler la commande `gotoAndStop`, qui doit faire avancer l'animation jusqu'à l'image `gameover`, nous passons ces deux valeurs à la racine :

```
MovieClip(root).gameScore = gameScore;
MovieClip(root).gameTime = clockTime(gameTime);
```

Vous remarquerez que nous passons le score sous forme de valeur brute, alors que nous faisons passer le temps par la fonction `clockTime` afin qu'il s'agisse d'une chaîne avec un signe deux-points et des secondes à deux chiffres.

Au niveau racine, nous devons définir ces nouvelles variables, qui utilisent les mêmes noms que les variables de jeu : `gameTime` et `gameScore`. J'ai ajouté ce code à la première image :

```
var gameScore:int;
var gameTime:String;
```

Ensuite, dans l'image `gameover`, nous utilisons ces variables pour placer des valeurs dans de nouveaux champs texte :

```
showScore.text = "Score: "+String(gameScore);
showTime.text = "Time: "+gameTime;
```

Il n'est pas nécessaire d'utiliser du code pour créer les champs texte dynamiques `showScore` et `showTime` ; cela peut se faire sur la scène avec les outils d'édition de Flash. La Figure 3.15 montre à quoi ressemble maintenant l'écran `gameover` lorsqu'une partie est terminée.

Figure 3.15
L'écran de fin de partie avec le score et le temps écoulé.

Pour simplifier les choses, nous avons ici inclus les chaînes `"Score: "` et `"Time: "` avec les champs `Score` et `Time`. Il aurait cependant été plus rigoureux d'inclure les mots `Score` et `Time` sous forme de texte statique ou d'image à l'écran et de n'inclure que le score et le temps eux-mêmes à l'intérieur des champs. Dans le cas présent, il est indispensable de faire passer la variable `gameScore` dans la fonction `String` (car la propriété `.text` d'un champ texte doit être une chaîne). En conservant simplement `gameScore`, vous tenteriez de définir une chaîne par un entier et provoqueriez un message d'erreur.

Voilà tout pour **MatchingGame9.fla** et **MatchingGameObject9.fla**. Nous disposons maintenant d'un jeu avec un écran d'introduction et un écran de fin de partie. Il tient le registre du score, enregistre le temps écoulé et affiche ces indications en fin de partie. Il permet également au joueur de lancer une nouvelle partie.

Nous allons maintenant finaliser le jeu en ajoutant une variété d'effets spéciaux, comme une animation pour le retournement des cartes, une limite de temps pour la visualisation des cartes et des effets sonores.

Ajouter des effets

Qu'ils semblent loin, les premiers temps des jeux sur le Web où la simple idée de pouvoir s'amuser à un jeu dans une page semblait déjà un exploit ! Il faut aujourd'hui s'employer à créer des jeux de qualité pour espérer attirer des visiteurs, et ce souci de perfection peut tenir à de simples touches de finition comme des animations et des sons.

Essayons de soigner notre jeu de Memory à l'aide de quelques effets spéciaux. S'ils ne changeront pas le jeu lui-même, ils contribueront à le rendre plus intéressant pour les joueurs.

Retournements de cartes animés

Puisque nous retournons sans cesse des cartes virtuelles, il est assez cohérent de vouloir représenter ce mouvement de retournement par une animation. Vous pourriez le faire avec une série d'images à l'intérieur d'un clip mais, puisqu'il s'agit ici d'apprendre à programmer, faisons-le en ActionScript.

Il est assez difficile ici d'utiliser une animation dans le scénario au lieu d'une animation ActionScript, par la nature même des cartes. Le but est d'animer non pas dix-huit cartes mais une seule. Il serait donc plus judicieux de placer les faces des cartes dans un autre clip et de changer l'image de ce clip imbriqué au lieu du clip `Card` *principal. Les images 2 et suivantes du clip* `Card` *peuvent alors être utilisées pour une séquence animée présentant un retournement de carte. Tout cela n'est pas simple à visualiser à moins d'être habitué à créer des animations Flash.*

Puisque cette animation affecte les cartes et elles seules, il est judicieux de la placer à l'intérieur de la classe `Card`. Nous ne possédons cependant pas de classe `Card`. Au début de ce chapitre, nous avons choisi de ne pas utiliser de classe `Card` et de permettre simplement à Flash de lui attribuer une classe par défaut.

Il est cette fois temps de créer une classe `Card`. Si nous créons un fichier **Card.as**, il sera cependant utilisé par tous les objets `Card` qui se trouvent dans le dossier. Nous avons déjà les animations **MatchingGame1.fla** à **MatchingGame9.fla** qui contiennent des objets `Card`. Afin de signifier clairement que nous souhaitons que seule l'animation **MatchingGame10.fla** utilise cette classe `Card`, nous changerons donc le nom du symbole et de la classe qu'il référence en choisissant `Card10`. Ensuite, nous créerons le fichier de classe ActionScript **Card10.as**.

Cette classe permettra de créer un retournement animé de la carte au lieu de la changer instantanément. Elle remplacera toutes les fonctions `gotoAndStop` dans la classe principale. Elle demandera aux cartes d'opérer le retournement avec `startFlip`. Elle passera également l'image que la carte doit afficher une fois le retournement effectué.

La classe `Card10` définira ensuite certaines variables, configurera un écouteur événementiel et enclenchera l'animation de la carte dans les dix images suivantes :

```
package {
    import flash.display.*;
    import flash.events.*;

    public dynamic class Card10 extends MovieClip {
        private var flipStep:uint;
        private var isFlipping:Boolean = false;
        private var flipToFrame:uint;

        // Commencer le retournement, mémoriser l'image à atteindre
        public function startFlip(flipToWhichFrame:uint) {
            isFlipping = true;
            flipStep = 10;
            flipToFrame = flipToWhichFrame;
            this.addEventListener(Event.ENTER_FRAME, flip);
        }

        // 10 étapes pour le retournement
        public function flip(event:Event) {
            flipStep--; // Étape suivante

            if (flipStep > 5) { // Première partie du retournement
                this.scaleX = .2*(flipStep-6);
            } else { // Deuxième partie du retournement
                this.scaleX = .2*(5-flipStep);
            }

            // Au milieu du retournement, passer à nouvelle image
            if (flipStep == 5) {
                gotoAndStop(flipToFrame);
            }

            // À la fin du retournement, arrêter l'animation
            if (flipStep == 0) {
                this.removeEventListener(Event.ENTER_FRAME, flip);
            }
        }
    }
}
```

La variable `flipStep` commence donc à `10` lorsque la fonction `startFlip` est appelée. Elle est ensuite réduite d'une unité à chaque image subséquente.

La propriété `scaleX` réduit ou étend la largeur d'un clip. La valeur par défaut est 1,0. La valeur 2,0 double sa largeur et la valeur 0,5 la réduit de moitié.

Si `flipStep` est comprise entre 6 et 10, la propriété `scaleX` de la carte se voit attribuer la valeur `.2*(flipStep-6)`, qui doit être 0,8, 0,6, 0,4, 0,2 et 0. Le clip se rétrécit donc à chaque étape.

Ensuite, lorsque `flipStep` est comprise entre 5 et 0, on utilise la nouvelle formule `.2*(5-flipStep)`. Le résultat devient donc 0, 0,2, 0,4, 0,6, 0,8, puis 1. Le clip retrouve ainsi sa taille normale.

Lors de la cinquième étape, la carte passe à la nouvelle image. Elle se rétrécit, jusqu'à se réduire à rien, passe à la nouvelle image puis s'agrandit de nouveau.

Pour réaliser cet effet, j'ai dû apporter une modification concernant la disposition des images dans le clip `Card`. Dans toutes les précédentes versions du jeu, nous avions placé le coin supérieur gauche des cartes au centre du clip. Pour que la modification de `scaleX` donne l'impression que la carte pivote autour de son centre, j'ai cependant dû centrer les images de carte dans chaque image sur le centre du clip. Pour observer cette différence, comparez les clips `Card` dans **MachingGame9.fla** et **MatchingGame10.fla**. La Figure 3.16 montre le résultat lors de l'édition des clips.

Lors de la dernière étape, l'écouteur événementiel est complètement supprimé.

L'intéressant avec cette classe tient à ce qu'elle fonctionne tout aussi bien lorsque la carte est recouverte (retournée face en bas) en allant à l'image 1.

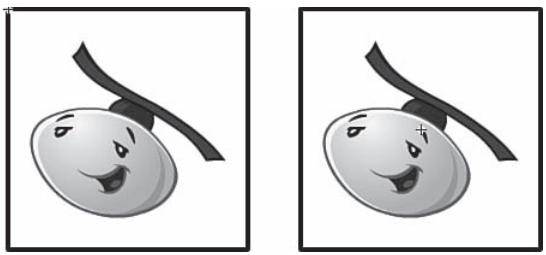


Figure 3.16
Le côté gauche fait apparaître le point d'alignement du clip en haut à gauche, comme dans les neuf premières animations d'exemple du chapitre. À doite, le clip est centré, comme dans l'exemple final.

Examinez **MatchingGameObject10.as** et remarquez comme tous les appels `gotoAndStop` ont été remplacés par `startFlip`. Grâce à cette modification, nous créons non seulement une animation de retournement, mais donnons également à la classe `Card` un meilleur contrôle sur elle-même. Idéalement, vous pourriez même donner aux cartes un contrôle complet sur elles-mêmes en attribuant plus de fonctions à la classe **Card10.as**, comme celles qui définissent l'emplacement des cartes au début du jeu.

Temps d'affichage limité des cartes

L'un des ajouts intéressants à ce jeu consiste à retourner automatiquement les paires de cartes non correspondantes une fois que le joueur les a regardées suffisamment longtemps. Par exemple, supposons que le joueur choisisse deux cartes. Elles ne forment pas une paire et restent donc affichées pour que le joueur les observe. Après deux secondes, les cartes se retournent, même si le joueur n'a pas commencé à sélectionner une autre paire.

Pour réaliser cela, nous utiliserons un `Timer`. Cet objet est très utile pour ajouter ce type de fonction. Pour commencer, nous devons importer la classe `Timer` dans notre classe principale :

```
import flash.utils.Timer;
```

Ensuite, nous devons créer une variable de minuteur au début de la classe :

```
private var flipBackTimer:Timer;
```

Plus loin dans la fonction `clickCard`, nous devons ajouter du code juste après que le joueur a choisi la seconde carte, qu'il n'a pas obtenu de correspondance et que son score a été diminué. Le code de `Timer` suivant configure un nouveau minuteur qui appelle simplement une fonction au bout de 2 secondes :

```
flipBackTimer = new Timer(2000,1);
flipBackTimer.addEventListener(TimerEvent.TIMER_COMPLETE,returnCards);
flipBackTimer.start();
```

L'événement `TimerEvent.TIMER_COMPLETE` se déclenche lorsqu'un minuteur a fini. Le plus souvent, le `Timer` boucle un certain nombre de fois, en déclenchant un événement `TimerEvent.TIMER` à chaque fois. Lors du dernier événement, il déclenche également l'événement `TimerEvent.TIMER_COMPLETE`. Comme nous ne souhaitons délencher qu'un seul événement à un moment ultérieur donné, nous fixons le nombre d'événements `Timer` à un, puis recherchons `TimerEvent.TIMER_COMPLETE`.

Après 2 secondes, la fonction `returnCards` est appelée. Il s'agit d'une nouvelle fonction qui procède comme la dernière partie de l'ancienne fonction `clickCard`. Elle retourne la première et la seconde sélection face vers le bas (masque les cartes) puis positionne les valeurs `firstCard` et `secondCard` à `null`. Elle supprime également l'écouteur :

```
public function returnCards(event:TimerEvent) {
    firstCard.startFlip(1);
    secondCard.startFlip(1);
    firstCard = null;
    secondCard = null;
    flipBackTimer.removeEventListener(TimerEvent.TIMER_COMPLETE,returnCards);
 }
```

La fonction `returnCards` duplique le code qui se trouvait auparavant dans `clickCard`. Dans **MatchingGameObject10.as**, j'ai donc remplacé ce code dupliqué dans `clickCard` par un simple appel à `returnCards`. C'est ainsi un seul et même emplacement dans notre code qui se charge de retourner les paires de cartes pour les masquer.

Comme `returnCards` requiert un paramètre d'événement, nous devons lui passer ce paramètre dans `returnCards` que nous ayons quelque chose à passer ou non. L'appel à l'intérieur de `clickCard` passe donc simplement la valeur `null` :

```
returnCards(null);
```

À présent, si vous exécutez l'animation et retournez deux cartes, elles se retournent automatiquement au bout de 2 secondes.

Comme nous avons une commande `removeEventListener` dans la fonction `returnCards`, l'écouteur est supprimé même si la fonction `returnCards` est déclenchée par le joueur qui retourne une autre carte. Sans cela, le joueur retournerait une nouvelle carte, les deux premières seraient masquées et l'événement serait déclenché après 2 secondes alors même que les deux cartes sont déjà cachées.

Effets sonores

Un jeu ne saurait être considéré comme complet s'il n'émet pas de son. ActionScript 3.0 permet aisément d'ajouter des effets sonores, bien que cette opération requière quelques étapes.

La première consiste à importer vos sons. Pour ma part, j'ai créé trois sons et souhaite les importer chacun dans la bibliothèque :

`FirstCardSound.aiff`

`MissSound.aiff`

`MatchSound.aiff`

Une fois les sons importés, il convient de modifier leurs propriétés. Nommez-les tous en utilisant leur nom de fichier sans l'extension .aiff. Cochez en outre l'option Exporter pour ActionScript et donnez-leur le même nom de classe que le nom de symbole. La Figure 3.17 présente la boîte de dialogue Propriétés de l'un des sons.

Ensuite, configurons la classe principale du jeu de manière à lire les sons au bon moment. Pour commencer, nous devons importer deux nouvelles classes afin de pouvoir utiliser du son :

```
import flash.media.Sound;
import flash.media.SoundChannel;
```

Ensuite, nous créons des variables de classe pour contenir des références à ces sons :

```
var theFirstCardSound:FirstCardSound = new FirstCardSound();
var theMissSound:MissSound = new MissSound();
var theMatchSound:MatchSound = new MatchSound();
```

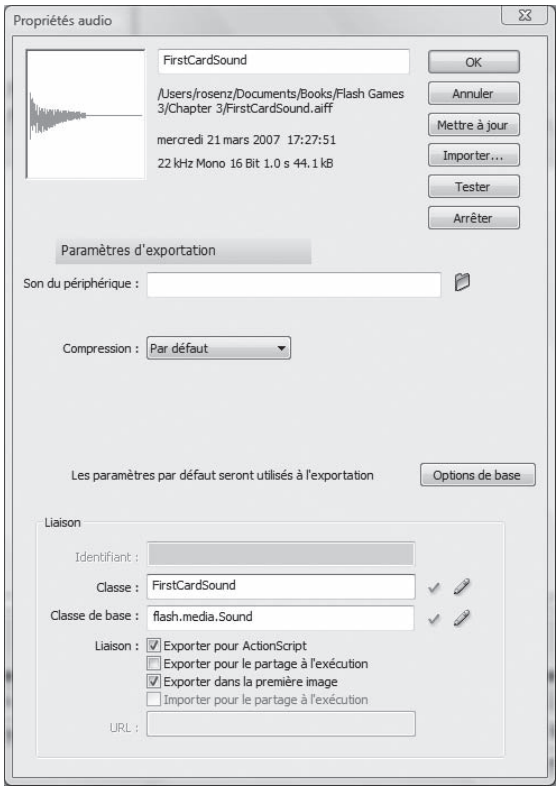


Figure 3.17
Chaque son est une classe et devient accessible dans le code ActionScript par le nom de sa classe.

Pour ma part, je préfère charger une seule et même fonction de procéder à la lecture de tous mes sons. Appelons cette fonction `playSound` et ajoutons-la à la fin de la classe :

```
public function playSound(soundObject:Object) {
    var channel:SoundChannel = soundObject.play();
}
```

À présent, lorsque nous souhaitons lire un son, il nous suffit d'appeler `playSound` avec la variable du son à utiliser, comme ceci :

```
playSound(theFirstCardSound);
```

Dans **MatchingGameObject10.as**, j'ai ajouté `playSound(theFirstCardSound)` lorsque le joueur clique sur la première carte et lorsqu'il clique sur une carte alors que deux cartes non correspondantes sont déjà retournées. J'ai ajouté `playSound(theMissSound)` lorsque la seconde carte est retournée et qu'il n'y a pas de correspondance. J'ai ajouté `playSound(theMatchSound)` lorsque la seconde carte est retournée et qu'une correspondance est trouvée.

Voilà tout ce qu'il faut pour ajouter des effets sonores à notre jeu.

À ce stade, il peut être intéressant de passer en revue vos paramètres de publication afin de choisir les réglages pour la compression des sons. Vous pouvez sinon définir la compression des sons individuellement pour chaque son dans les propriétés du symbole. Dans un cas comme dans l'autre, il est sans doute préférable d'utiliser un réglage à forte compression, comme MP3 16Mbps, car il ne s'agit que de simples effets sonores.

Modifier le jeu

Il nous reste quelques autres petites modifications avant de terminer ce jeu.

Tout d'abord, lorsque nous avons recentré toutes les cartes, nous avons provoqué un décalage horizontal et vertical pour le placement des cartes. Il faut ajuster cela :

```
private static const boardOffsetX:Number = 145;
private static const boardOffsetY:Number = 70;
```

Comment ai-je calculé ces valeurs ? Voici le raisonnement :

- La scène fait 550 pixels de largeur. Il y a six cartes en largeur espacées de 52 pixels. Cela fait 550 – 6 × 52 pour l'espace total restant à gauche et à droite. Je divise le tout par deux pour obtenir l'espace à droite. Les cartes sont cependant centrées sur 0,0. Je dois donc soustraire la moitié de la largeur d'une carte, soit 26. Ainsi, (550 – 6 × 52) / 2 – 26 = 145.
- Idem pour le décalage vertical : (400 – 6 × 52) / 2 – 26 = 70.

Reste encore à considérer le curseur. Lorsque le joueur se positionne pour cliquer sur une carte, aucun curseur spécial ne lui signale qu'il peut cliquer à cet endroit. On peut aisément remédier à cela en positionnant la propriété `buttonMode` de chaque carte au moment de la créer.

```
c.buttonMode = true;
```

Un curseur en forme de main apparaît maintenant lorsque l'utilisateur survole les cartes. C'est déjà le cas pour les boutons Play et Play Again car il s'agit de symboles de bouton.

J'ai enfin opéré une dernière modification en augmentant la cadence d'images de l'animation afin de passer de 12 ips à 60 ips. Vous pouvez le faire en choisissant Modification > Document et en paramétrant les propriétés du document de l'animation.

À 60 ips, les animations de retournement des cartes sont bien plus fluides. Et, grâce au moteur ActionScript 3.0 ultrarapide, les ordinateurs lents peuvent exécuter ce jeu avec cette cadence d'images élevée.

Voilà tout pour notre jeu de Memory, dont la version finale utilise les fichiers suivants :

MatchingGame10.fla

MatchingGameObject10.as

Card10.as

4

Jeux cérébraux : Simon et déduction

Au sommaire de ce chapitre

- Tableaux et objets de données
- Jeu de Simon
- Jeu de déduction

Au chapitre précédent, nous avons examiné un jeu qui intégrait une unique configuration de grille de jeu et dont la partie se terminait une fois la grille entièrement effacée. Bien des jeux proposent cependant plusieurs configurations possibles. Ils créent une situation pour le joueur, celui-ci agit, puis la situation suivante est mise en place. Ces jeux peuvent être considérés comme des jeux à tours.

Dans ce chapitre, nous allons examiner deux jeux de ce type : le jeu de Simon et le jeu de déduction. Le premier demande au joueur d'observer et de répéter une séquence. À chaque tour, la séquence s'allonge, jusqu'à ce que le joueur ne suive plus. Le second demande au joueur de deviner une séquence, en enchaînant des tours et en se servant des indications fournies à chaque étape pour tenter de se rapprocher de la solution au tour suivant.

La configuration simple utilisée au précédent chapitre ne fonctionne pas pour ces jeux. Nous devons utiliser des tableaux et des objets de données pour stocker des informations concernant le jeu et utiliser ces objets de données pour déterminer le résultat de chaque tour du joueur.

Tableaux et objets de données

Les jeux que nous allons créer dans ce chapitre requièrent que l'on stocke des informations concernant la partie et les déplacements du joueur. Pour cela, nous allons utiliser ce que les informaticiens appellent des *structures de données*.

Les structures de données désignent tout simplement des méthodes pour le stockage de groupes d'informations. La structure de données la plus simple est le tableau, qui stocke une liste d'informations. ActionScript propose également des objets de données qui permettent de stocker des informations étiquetées. Vous pouvez imbriquer cette structure dans la première et créer ainsi un tableau d'objets de données.

Tableaux

Un tableau est une liste de valeurs. Par exemple, si nous souhaitons qu'un joueur puisse choisir parmi une liste de personnages au début d'une partie, nous pouvons stocker cette liste de la manière suivante :

```
var characterTypes:Array = new Array();
characterTypes = ["Warrior", "Rogue", "Wizard", "Cleric"];
```

Nous pourrions également utiliser la commande `push` pour ajouter des éléments au tableau. Le code suivant produit ainsi le même résultat que les lignes précédentes :

```
var characterTypes:Array = new Array();
characterTypes.push("Warrior");
characterTypes.push("Rogue");
characterTypes.push("Wizard");
characterTypes.push("Cleric");
```

Cet exemple crée un tableau de chaînes. Les tableaux peuvent cependant contenir n'importe quel type de valeur, comme des nombres ou même des objets d'affichage tels que des sprites ou des clips.

Les tableaux peuvent non seulement contenir des valeurs de n'importe quel type, mais également combiner ces différents types. Il est ainsi possible de créer un tableau comme le suivant : `[7, "Hello"]`.

L'un des usages courants des tableaux dans les jeux consiste à stocker les clips et les sprites au fur et à mesure qu'ils sont créés. Par exemple, au Chapitre 3, nous avons créé une grille de paires de cartes. Pour pouvoir y accéder facilement, nous aurions pu stocker une référence à chaque `Card` dans un tableau.

Ce tableau aurait pu être créé de la manière suivante si nous avions voulu créer dix cartes :

```
var cards:Array = new Array();
for(var i:uint=0;i<10;i++) {
    var thisCard:Card = new Card();
    cards.push(thisCard);
}
```

Il y a plusieurs avantages à ce que les pièces de votre jeu se trouvent dans un tableau. Par exemple, il est aisé de les parcourir en boucle puis de vérifier chaque pièce afin de retrouver des correspondances ou de détecter des collisions.

Vous pouvez également imbriquer des tableaux afin de créer des tableaux de tableaux. Cela se révèle particulièrement utile pour les grilles de pièces de jeu comme pour le Chapitre 3. Par exemple, une grille de morpion pourrait être représentée comme ceci : `[["X","O","O"], ["O","X","O"], ["X","O","X"]]`.

Vous pouvez ajouter de nouveaux éléments aux tableaux, leur retirer des éléments, les trier et faire porter une recherche sur leur contenu. Le Tableau 4.1 liste certaines des fonctions de tableau les plus courantes.

Tableau 4.1 : Fonctions de tableau courantes

Fonction	*Exemple*	*Description*
`push`	`myArray.push("Wizard")`	Ajoute une valeur à la fin d'un tableau
`pop`	`myArray.pop()`	Supprime la dernière valeur d'un tableau et la retourne
`unshift`	`myArray.unshift("Wizard")`	Ajoute une valeur au début d'un tableau

Tableau 4.1 : Fonctions de tableau courantes *(Suite)*

Fonction	*Exemple*	*Description*
`shift`	`myArray.shift("Wizard")`	Supprime la première valeur d'un tableau et la retourne
`splice`	`myArray.splice(7,2,"Wizard","Bard")`	Supprime les éléments d'un emplacement du tableau et y insère de nouveaux éléments
`indexOf`	`myArray.indexOf("Rogue")`	Retourne l'emplacement d'un élément ou `-1` si l'élément n'est pas trouvé
`sort`	`myArray.sort()`	Trie un tableau

Les tableaux sont des structures de données courantes et indispensables dans les jeux. En fait, l'ensemble des structures de données dont il sera question dans cette section utilise des tableaux pour convertir un unique élément de données en une liste d'éléments de données.

Objets de données

Les tableaux font merveille lorsqu'il s'agit de stocker des listes de valeurs uniques, mais que faire lorsque l'on souhaite regrouper des valeurs les unes avec les autres ? Supposons que, dans un jeu d'aventure, vous souhaitiez mémoriser des types de personnages, des niveaux et des points de santé dans un groupe. Supposons ainsi qu'un personnage à l'écran soit un "Guerrier", au niveau 15, avec un niveau de santé compris entre 0,0 et 1,0. Vous pourriez utiliser un objet de données pour stocker ensemble ces trois éléments d'information.

Dans certains autres langages de programmation, l'équivalent des objets de données correspond en fait aux tableaux associatifs. Comme les objets de données, les tableaux associatifs sont des listes d'éléments qui incluent une étiquette (une clé) et une valeur. Vous pouvez en réalité utiliser les tableaux standard de cette manière dans ActionScript, mais ils ne sont pas aussi flexibles que les objets de données.

Pour créer un objet de données, vous pouvez le définir comme étant de type `Object`, puis lui ajouter des propriétés avec la syntaxe à points :

```
var theCharacter:Object = new Object();
theCharacter.charType = "Warrior";
theCharacter.charLevel = 15;
theCharacter.charHealth = 0.8;
```

Si vous le souhaitez, vous pouvez créer cette même variable de la manière suivante :

```
var theCharacter:Object = {charType: "Warrior", charLevel: 15, charHealth: 0.8};
```

Les `Object` sont dynamiques, ce qui signifie que vous pouvez leur ajouter de nouvelles propriétés dès que vous le souhaitez et que ces propriétés peuvent être de n'importe quel type. Il n'est pas nécessaire de déclarer les variables à l'intérieur d'un `Object` ; vous n'avez qu'à leur attribuer une valeur comme dans l'exemple précédent.

Les objets de données et les tableaux interagissent très bien ensemble. Par exemple, vous pouvez créer un tableau de personnages comme le précédent.

Les objets de données en ActionScript ne diffèrent pas vraiment des objets normaux. Vous pouvez même attribuer une fonction à un objet de données. Par exemple, si vous avez un objet avec les propriétés `prenom` et `nom`, vous pouvez créer une fonction `nomcomplet()` qui retourne `prenom+" "+nom`.

Tableaux d'objets de données

À partir de maintenant, nous utiliserons dans presque chaque jeu des tableaux d'objets de données pour tenir le registre des éléments du jeu. Nous pourrons ainsi stocker les sprites ou les clips eux-mêmes en plus des données les concernant.

Par exemple, un objet de données pourrait ressembler à ceci :

```
var thisCard:Object = new Object();
thisCard.cardobject = new Card();
thisCard.cardface = 7;
thisCard.cardrow = 4;
thisCard.cardcolumn = 2;
```

À présent, imaginons un tableau entier rempli de ces objets. Dans le jeu de Memory du Chapitre 3, nous aurions pu placer toutes les cartes dans des objets de cette sorte.

Sinon imaginons un ensemble complet d'éléments à l'écran, comme dans un jeu d'arcade. Un tableau d'objets stockerait des informations les concernant, comme la vitesse, le comportement, l'emplacement, etc.

Il existe un autre type d'objet, appelé `Dictionary`. Les dictionnaires peuvent être utilisés comme les `Object`, à ceci près qu'il est possible d'utiliser n'importe quelle valeur comme clé, comme des sprites, des clips, d'autres objets, enfin, en somme, pratiquement n'importe quoi.

Les structures de données comme les tableaux et les objets de données sont essentielles dans tous les jeux à l'exception des plus simples. Utilisons-les dans deux exemples de jeu complets.

Jeu de Simon

Codes sources

http://flashgameu.com

A3GPU04_MemoryGame.zip

Le jeu de Simon est un autre jeu simple pour les adultes et les enfants. C'est un jeu plus récent que le Memory, lequel peut être joué sans équipement technologique.

Le jeu de Simon consiste à présenter une séquence d'images ou de sons que le joueur doit tenter de reproduire. En général, la séquence commence par un élément puis en ajoute un nouveau à chaque tour. Le joueur doit ainsi répéter une séquence à un élément, puis deux, puis trois, etc., comme A, puis AD, puis ADC, puis ADCB, puis ADCBD, etc. En fin de compte, la séquence devient si longue que le joueur finit par faire une erreur et que la partie se termine.

La version la plus connue du jeu de Simon a été popularisée par le jouet du même nom commercialisé dans le courant de l'année 1978. Il a été créé par Ralph Baer, considéré comme l'un des pères des jeux sur ordinateur. Ralph Baer a créé le jeu original Magnavox Odyssey, premier jeu de console familial. En 2005, il s'est vu attribuer la Médaille nationale de la technologie pour son rôle de création dans l'industrie du jeu vidéo.

Préparer l'animation

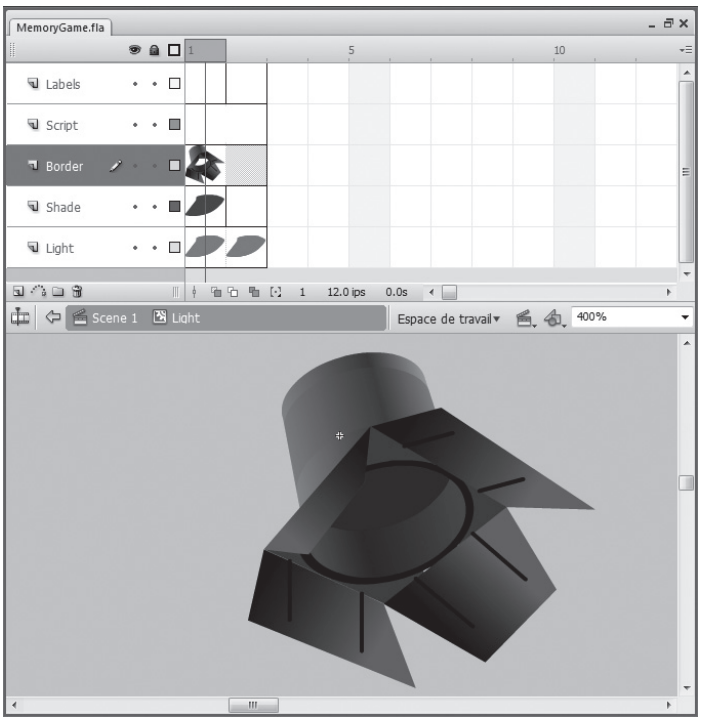
Conformément au style ActionScript 3.0, nous allons créer tous les éléments du jeu dans notre code. Cela implique de commencer par un scénario principal vide mais pas une bibliothèque vide. La bibliothèque doit contenir au moins le clip pour les pièces du jeu, qui seront des projecteurs dans cet exemple.

Nous aurons cinq projecteurs, mais tous seront contenus dans un même clip. En outre, il doit exister deux versions de chaque projecteur : allumé et éteint.

Le clip `Lights` lui-même (voir Figure 4.1) inclut deux images qui contiennent un autre clip, `LightColors`. Dans la première image de `Lights`, un cache masque le clip `LightColors` et estompe sa couleur dans le calque `Shade`. Il s'agit d'un cache noir dont la propriété alpha est fixée à 75 %,

ce qui signifie que seulement 25 % de la couleur sous-jacente transparaît. La première image correspond donc à une couleur atténuée qui représente l'état éteint des projecteurs. La seconde image ne contient plus ce cache et correspond donc à l'état allumé.

Figure 4.1
Le scénario du clip `Lights` contient deux images : éteint et allumé. Le clip est ici présenté avec le mode Aperçu, auquel vous pouvez accéder depuis le menu déroulant situé en haut à droite du scénario.



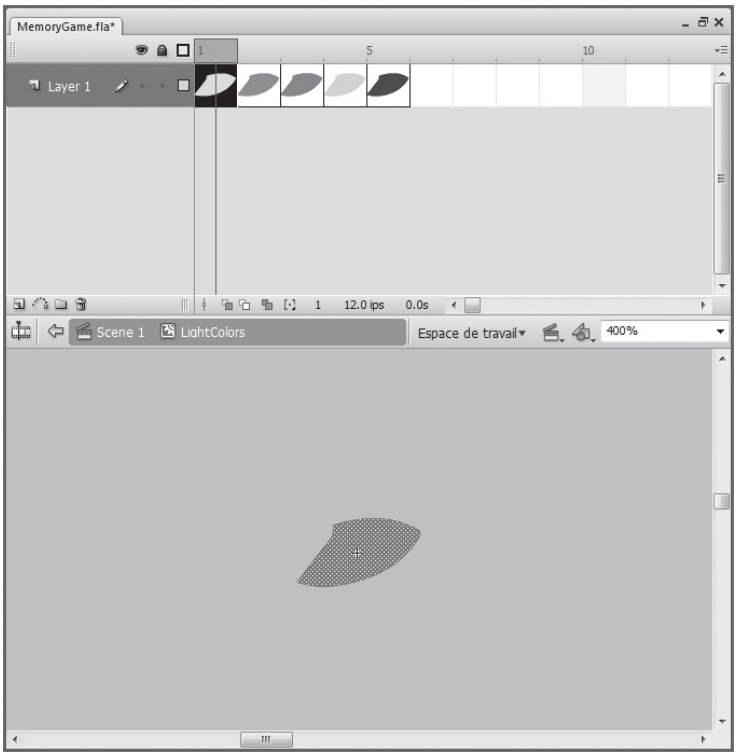
Il n'existe ni bon ni mauvais moyen de créer des éléments de jeu comme les projecteurs de ce jeu de Simon. Vous pouvez avoir un clip pour chaque projecteur ou pour chaque état de projecteur. Vous pourriez aussi placer les dix variantes (cinq projecteurs et deux états possibles) dans un scénario à dix images. Parfois, ces choix ne sont qu'affaire de goût. Si vous êtes programmeur et que vous travailliez avec un artiste sur un jeu, l'approche peut être adaptée pour faciliter la création des images par l'artiste.

Le clip `LightColors` (voir Figure 4.2) contient cinq images qui affichent toutes une couleur différente.

Le clip `LightColors` est nommé `lightColors` avec un l minuscule. Pour modifier la couleur d'un projecteur, il suffit donc d'utiliser `lightColors.gotoAndStop` avec le numéro d'image.

Nous nommerons l'animation **MemoryGame.fla** et le fichier ActionScript **MemoryGame.as**. Cela signifie que la classe du document doit être MemoryGame dans l'inspecteur des propriétés, comme nous l'avons fait pour le jeu de Memory au Chapitre 3.

Figure 4.2
Le scénario du clip `LightColors` *contient une couleur par image.*



Stratégie de programmation

L'animation commence avec rien, puis ActionScript crée tous les éléments à l'écran. Nous devons donc créer les cinq projecteurs et leur attribuer chacun une couleur. Ensuite, nous devons créer deux champs texte : l'un pour indiquer aux joueurs s'ils doivent observer la séquence ou tenter de la reproduire et l'autre pour leur faire savoir le nombre de projecteurs actuellement impliqués dans la séquence.

Il existe de nombreuses autres solutions qu'utiliser deux champs texte pour afficher des informations. Par exemple, le nombre d'éléments dans la séquence pourrait apparaître dans un cercle ou un rectangle sur un côté. Le texte "Watch and listen" (observez et écoutez) et le texte "Repeat" (reproduisez) pourraient au lieu de cela être des symboles qui s'allument comme des feux vert et rouge de circulation. Les champs texte sont simplement un moyen pratique pour ne pas se soucier de ces éléments et pour se concentrer ici sur le code de la logique du jeu.

Les clips `Light` seront stockés dans un tableau. Il y a cinq projecteurs, soit cinq éléments dans le tableau. Ce tableau nous permettra plus facilement de nous référer aux clips lorsque nous aurons besoin de les allumer ou de les éteindre.

Nous stockerons également la séquence dans un tableau. Elle commencera sous la forme d'un tableau vide et nous y ajouterons un projecteur aléatoire à chaque tour.

Une fois la lecture d'une séquence terminée, nous dupliquerons le tableau de la séquence. À mesure que le joueur cliquera sur les projecteurs pour reproduire la séquence, nous supprimerons un élément du devant du tableau à chaque clic. Si cet élément de la séquence correspond au clic, nous en déduirons que le joueur a fait le bon choix.

Nous utiliserons également des objets `Timer` dans ce jeu. Pour lire la séquence, un minuteur appellera une fonction à chaque seconde afin d'allumer un projecteur. Ensuite, un second minuteur déclenchera une fonction pour éteindre le projecteur au bout d'une autre demi-seconde.

Définition de classe

Le fichier **MemoryGame.as** contiendra le code de ce jeu. N'oubliez pas de l'associer à **MemoryGame.fla** en définissant la Classe du document dans l'inspecteur des propriétés.

Pour commencer le code, nous déclarerons le paquetage et la classe. Il faut importer quelques classes Flash. En plus de la classe `flash.display.*` pour afficher les clips, nous avons besoin de la classe `flash.events.*` pour les clics de souris, de la classe `flash.text.*` pour l'affichage du texte et de `flash.utils.Timer` pour les minuteurs. Les classes `flash.media.Sound` et `flash.media.SoundChannel` sont requises pour lire les sons qui accompagnent les projecteurs. La classe `flash.net.URLRequest` est requise pour charger les sons depuis des fichiers externes :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
    import flash.utils.Timer;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;
```

Comment connaissais-je les noms des classes à importer au début de ce code ? Simple : je les ai recherchés dans les pages de l'aide Flash. Par exemple, pour trouver ce dont j'avais besoin pour un champ texte, j'ai consulté `TextField` et la définition m'a indiqué qu'il fallait `flash.text.*`. En fait, au lieu d'examiner la page de documentation, je saute en général directement à la fin de la page pour examiner le code d'exemple. La commande `import` est très facile à retrouver de cette manière.

La définition de classe inclut plusieurs déclarations de variables. La seule constante que nous utiliserons est le nombre de projecteurs dans le jeu (ici, cinq) :

```
public class MemoryGame extends Sprite {
    static const numLights:uint = 5;
```

Nous aurons trois tableaux principaux : l'un pour contenir des références aux cinq clips de projecteur, puis deux pour contenir la séquence de projecteurs. Le tableau `playOrder` s'étendra à chaque tour. Le tableau `repeatOrder` contiendra un duplicata du tableau `playOrder` lorsque le joueur reproduit la séquence. Il se réduira à mesure que le joueur cliquera sur les projecteurs et que les comparaisons seront effectuées avec chacun des projecteurs dans la séquence :

```
private var lights:Array; // Liste des objets projecteur
private var playOrder:Array; // Séquence croissante
private var repeatOrder:Array;
```

Il nous faut deux champs texte, l'un pour contenir un message à destination du joueur en haut de l'écran et un autre pour contenir la longueur de la séquence actuelle en bas de l'écran :

```
// Message texte
private var textMessage:TextField;
private var textScore:TextField;
```

Nous utiliserons deux minuteurs dans le jeu. Le premier allumera chaque projecteur dans la séquence au cours de sa lecture. Le second sera utilisé pour éteindre les projecteurs une demi-seconde plus tard :

```
// Minuteurs
private var lightTimer:Timer;
private var offTimer:Timer;
```

Parmi les autres variables requises figure `gameMode`, qui stockera la valeur `"play"` ou la valeur `"replay"` selon que le joueur observe la séquence ou tente de la reproduire. La variable `currentSelection` stockera une référence aux clips `Light`. Le tableau `soundList` contiendra des références aux cinq sons qui seront joués avec les projecteurs :

```
var gameMode:String; // Lecture (play) ou reproduction (replay)
var currentSelection:MovieClip = null;
var soundList:Array = new Array(); // Contient les sons
```

Voilà l'ensemble des variables que nous devons suivre. Nous aurions également pu inclure des constantes pour le positionnement du texte et des projecteurs, mais nous les coderons en dur afin de nous concentrer sur le fonctionnement du code.

Configurer le texte, les projecteurs et les sons

La fonction constructeur `MemoryGame` s'exécute dès que la classe est initialisée. Nous l'utiliserons pour configurer l'écran du jeu et charger les sons. La Figure 4.3 présente l'écran au tout début du jeu.

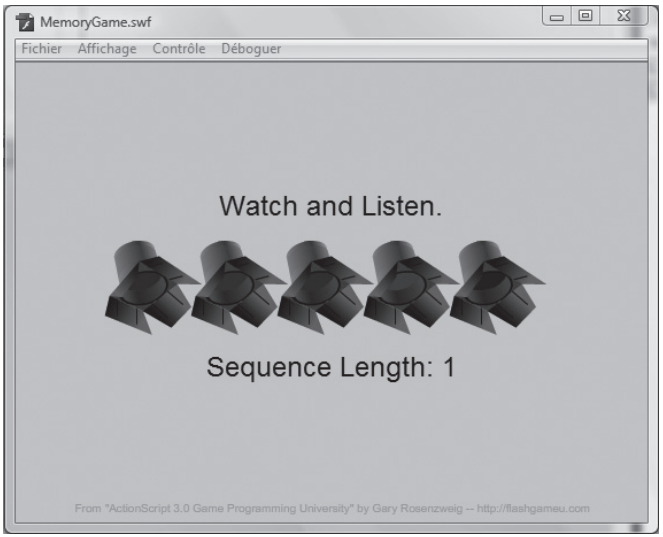


Figure 4.3
L'écran du jeu de Simon présente deux champs texte et cinq projecteurs.

Ajouter le texte

Avant de configurer l'un ou l'autre champ texte, nous allons créer un objet `TextFormat` temporaire et définir l'apparence que nous souhaitons donner à notre texte. Nous utiliserons cette variable temporaire lors de la création des deux champs texte et n'en aurons plus besoin ensuite. Il n'est donc pas nécessaire de définir la variable `textFormat` (notez l'utilisation du t minuscule) dans la classe principale, mais uniquement dans cette fonction :

```
public function MemoryGame() {
    // Mise en forme du texte
    var textFormat = new TextFormat();
    textFormat.font = "Arial";
    textFormat.size = 24;
    textFormat.align = "center";
```

Le champ texte du haut, qui doit être nommé `textMessage`, contiendra un message à l'intention du joueur lui indiquant s'il doit observer et écouter la séquence ou s'il doit cliquer sur les projecteurs afin de la reproduire.

Nous le placerons vers le haut de l'écran. Il fera 550 pixels de large, soit la largeur complète de l'écran. Comme `textFormat.align` vaut `"center"` et comme le champ texte couvre la largeur de l'écran, le texte doit être centré à l'écran.

Nous devons également positionner la propriété `selectable` du champ à `false`. Sans cela, le curseur se changera en un curseur de sélection de texte lorsque le joueur survolera le champ.

L'oubli qui consiste à ne pas attribuer la valeur `false` à la propriété `selectable` d'un champ texte fait partie des erreurs courantes. Par défaut, la propriété `selectable` vaut `true`, ce qui signifie que le curseur se transforme en un curseur d'édition de texte lorsque le joueur survole le champ. L'utilisateur peut alors sélectionner le texte mais, plus important encore, ne peut aisément cliquer sur les objets qui figurent sous le texte.

Pour finir, nous positionnons `defaultTextFormat` à notre objet `textFormat` afin de définir la police, la taille de police et l'alignement du texte pour le champ :

```
// Création du champ texte supérieur
textMessage = new TextField();
textMessage.width = 550;
textMessage.y = 110;
textMessage.selectable = false;
textMessage.defaultTextFormat = textFormat;
addChild(textMessage);
```

Le second champ texte affiche la longueur de la séquence courante afin que le joueur puisse évaluer son avancement. Il se trouvera vers le bas de l'écran :

```
// Création du champ texte inférieur
textScore = new TextField();
textScore.width = 550;
textScore.y = 250;
textMessage.selectable = false;
textScore.defaultTextFormat = textFormat;
addChild(textScore);
```

Charger les sons

Ensuite, nous allons charger les sons. Au Chapitre 3, nous avons utilisé des sons qui se trouvaient dans la bibliothèque de l'animation. ActionScript 3.0 ne rend pas cette méthode flexible parce que chaque son dans la bibliothèque doit être référencé comme son propre objet. Pour utiliser cinq sons, "note1" à "note5", il faudrait alors cinq objets séparés et des lignes de code distinctes pour chacun.

ActionScript 3.0 propose en fait un jeu de commandes bien plus robuste pour la lecture des fichiers son externes. Nous les utiliserons pour ce jeu. Pour cela, nous chargerons cinq fichiers son, "**note1.mp3**" à "**note5.mp3**", dans un tableau de sons.

Flash insiste sur le fait que les sons externes soient au format MP3. Le grand intérêt du MP3 tient à ce que vous pouvez véritablement contrôler la taille et la qualité d'un fichier avec votre logiciel d'édition audio. Vous pouvez donc créer des sons de petite taille et de faible qualité lorsqu'il convient de réduire le temps de téléchargement ou des sons volumineux de haute qualité lorsque cette fidélité est requise.

```
// Chargement des sons
soundList = new Array();
for(var i:uint=1;i<=5;i++) {
    var thisSound:Sound = new Sound();
    var req:URLRequest = new URLRequest("note"+i+".mp3");
    thisSound.load(req);
    soundList.push(thisSound);
}
```

Le `"note"+i+".mp3"` à l'intérieur de la fonction `URLRequest` construit une chaîne du type `"note1.mp3"`. Le symbole `+` concatène les chaînes et les autres éléments en une chaîne plus longue. Le résultat correspond donc à la concaténation de `"note"` plus la valeur de la variable `i`, plus `".mp3"`.

Ajouter les clips *Light*

Maintenant que nous avons des champs texte et des sons, il ne nous reste plus qu'à ajouter les projecteurs. Nous créerons cinq clips `Light` et les espacerons de manière à les centrer. Pour chaque objet `Light`, nous enverrons le clip `lightColors` intérieur à une image différente afin que chaque clip possède une couleur différente.

En plus d'ajouter les clips à la scène avec `addChild`, nous les ajouterons au tableau `lights` pour nous y référer par la suite. `addEventListener` permet aux clips de réagir aux clics de souris en appelant la fonction `clickLight`. Nous positionnerons également la propriété `buttonMode` de manière que le curseur se transforme en une main lorsque l'utilisateur survole le projecteur :

```
// Création des projecteurs
lights = new Array();
for(i=0;i<numLights;i++) {
    var thisLight:Light = new Light();
    thisLight.lightColors.gotoAndStop(i+1); // Afficher l'image appropriée
    thisLight.x = i*75+100; // Position
    thisLight.y = 175;
    thisLight.lightNum = i; // Mémoriser numéro de projecteur
    lights.push(thisLight); // Ajouter au tableau des projecteurs
    addChild(thisLight); // Ajouter à l'écran
    thisLight.addEventListener(MouseEvent.CLICK,clickLight);
    thisLight.buttonMode = true;
}
```

Tous les éléments écran ont été créés. Il est maintenant temps de commencer le jeu. Nous allons attribuer un nouveau tableau vide à `playOrder`, `gameMode`, pour "jouer" puis appeler `nextTurn` pour démarrer le premier projecteur dans la séquence :

```
        // Réinitialisation de la séquence, premier tour
        playOrder = new Array();
        gameMode = "play";
        nextTurn();
    }
```

Lire la séquence

La fonction `nextTurn` s'occupe de déclencher chaque séquence de lecture. Elle ajoute un projecteur aléatoire à la séquence, positionne le message en haut de l'écran en affichant "Watch and Listen" et déclenche le `lightTimer` qui affiche la séquence :

```
    // Ajoute un à la séquence et démarre
    public function nextTurn() {
        // Ajoute un nouveau projecteur à la séquence
        var r:uint = Math.floor(Math.random()*numLights);
        playOrder.push(r);

        // Affiche le texte
        textMessage.text = "Watch and Listen.";
        textScore.text = "Sequence Length: "+playOrder.length;

        // Configure les minuteurs pour afficher la séquence
        lightTimer = new Timer(1000,playOrder.length+1);
        lightTimer.addEventListener(TimerEvent.TIMER,lightSequence);

        // Lance le minuteur
        lightTimer.start();
    }
```

Lorsqu'une séquence commence à jouer, la fonction `lightSequence` est appelée à chaque seconde. L'événement est passé en paramètre. Le `currentTarget` de cet `event` est l'équivalent du `Timer`. L'objet `Timer` possède une propriété nommée `currentCount` qui retourne le nombre de fois où le minuteur s'est arrêté. Nous l'insérons dans `playStep` et pouvons l'utiliser pour déterminer le projecteur dans la séquence à allumer.

La fonction vérifie la valeur de `playStep` afin de déterminer s'il va s'agir du dernier arrêt du minuteur. Si c'est le cas, au lieu d'afficher un projecteur, elle commence la deuxième moitié d'un tour, lorsque le joueur doit répéter la séquence :

```
// Lire l'élément suivant de la séquence
public function lightSequence(event:TimerEvent) {
    // Où nous trouvons-nous dans la séquence ?
    var playStep:uint = event.currentTarget.currentCount-1;
    if (playStep < playOrder.length) { // Ce n'est pas la dernière étape
        lightOn(playOrder[playStep]);
    } else { // La séquence est terminée
        startPlayerRepeat();
    }
}
```

Allumer et éteindre les projecteurs

Lorsqu'il est temps pour le joueur de commencer à répéter la séquence, nous remplaçons le message texte par "Repeat" et attribuons à `gameMode` la valeur `"replay"`. Ensuite, nous créons une copie de la liste `playOrder` :

```
// Début de la répétition du joueur
public function startPlayerRepeat() {
    currentSelection = null;
    textMessage.text = "Repeat.";
    gameMode = "replay";
    repeatOrder = playOrder.concat();
}
```

Pour créer une copie d'un tableau, nous utilisons la fonction `concat`. Bien qu'elle soit destinée à créer un nouveau tableau à partir de plusieurs tableaux, elle fonctionne aussi bien pour créer un nouveau tableau à partir d'un seul autre tableau. Pourquoi procéder de cette manière au lieu de créer simplement un nouveau tableau et de le définir comme étant égal au premier ? Si nous posons une égalité entre les deux tableaux, ces tableaux seront littéralement identiques. En modifiant l'un, nous modifierons l'autre. Pour notre part, nous souhaitons créer un second tableau qui soit une copie du premier, de manière que les modifications du second n'affectent pas le premier. Voilà ce que nous permet de faire la fonction `concat`.

Les deux fonctions suivantes allument et éteignent un projecteur (`Light`). Nous passons le nombre de projecteurs dans la fonction. L'allumage d'un projecteur requiert simplement d'utiliser `gotoAndStop(2)` pour envoyer le clip `Light` à la seconde image qui ne contient pas le cache recouvrant la couleur.

Au lieu d'utiliser les numéros d'image 1 et 2, nous aurions également pu intituler les images "on" et "off" et utiliser des noms d'étiquette d'image. Cette méthode serait particulièrement utile dans des jeux où interviennent plus de deux modes par clip.

Nous lirons également le son associé au projecteur, mais en utilisant une référence au son dans le tableau `soundList` que nous avons créé.

`lightOn` fait encore une dernière chose : créer et démarrer le minuteur `offTimer`. Celui-ci ne se déclenche qu'une seule fois, 500 millisecondes après l'allumage du projecteur :

```
// Allumage du projecteur et configuration du minuteur pour l'éteindre
public function lightOn(newLight) {
    soundList[newLight].play(); // Lecture du son
    currentSelection = lights[newLight];
    currentSelection.gotoAndStop(2); // Allumage du projecteur
    offTimer = new Timer(500,1); // Rappel pour l'éteindre
    offTimer.addEventListener(TimerEvent.TIMER_COMPLETE,lightOff);
    offTimer.start();
}
```

La fonction `lightOff` renvoie ensuite le clip `Light` à la première image. C'est là qu'il s'avère pratique de stocker une référence au clip `Light` dans `currentSelection`.

Cette fonction demande également au minuteur `offTimer` de s'arrêter. Pourtant, si `offTimer` ne se déclenche qu'une seule fois, à quoi cela sert-il ? En réalité, si `offTimer` ne se déclenche qu'une seule fois, `lightOff` pourrait être appelée deux fois. Cela se produit si le joueur répète la séquence et clique suffisamment rapidement sur les projecteurs pour qu'ils s'éteignent avant que les 500 millisecondes expirent. Dans ce cas, `lightOff` est appelée une fois pour le clic de souris, puis une nouvelle fois lorsque le minuteur `lightOff` s'arrête. En émettant une commande `offTimer.stop()`, nous parvenons ainsi à arrêter ce second appel à `lightOff` :

```
// Éteindre le projecteur si toujours allumé
public function lightOff(event:TimerEvent) {
    if (currentSelection != null) {
        currentSelection.gotoAndStop(1);
        currentSelection = null;
        offTimer.stop();
    }
}
```

Accepter et vérifier l'entrée de l'utilisateur

La dernière fonction requise pour le jeu est appelée lorsque le joueur clique sur un projecteur (`Light`) pendant la répétition de la séquence.

Elle commence par une vérification du mode de jeu (`gameMode`) afin de s'assurer que `playMode` vaut `"replay"`. Sinon le joueur ne doit pas cliquer sur les projecteurs et la commande `return` est utilisée pour sortir de la fonction.

Si `return` est généralement utilisée pour renvoyer une valeur à partir d'une fonction, elle peut également être utilisée pour terminer une fonction qui n'est pas censée retourner de valeur du tout. Dans un tel cas, il suffit d'une simple commande `return` seule. Si la fonction est supposée retourner une valeur, `return` doit cependant être suivie par la valeur.

En supposant que cela ne se produise pas, la fonction `lightOff` est appelée pour éteindre le projecteur précédent s'il ne l'est pas déjà.

Une comparaison est ensuite opérée. Le tableau `repeatOrder` contient un duplicata du tableau `playOrder`. Nous utilisons la commande `shift` pour retirer le premier élément du tableau `repeatOrder` et le comparer à la propriété `lightNum` du projecteur sur lequel le joueur a cliqué.

Rappelez-vous que `shift` retire un élément du devant du tableau, alors que `pop` le retire à l'autre extrémité. Si vous souhaitez simplement tester le premier élément d'un tableau et non l'enlever, vous pouvez utiliser `monTableau[0]`. De la même manière, vous pouvez utiliser `monTableau[monTableau.length-1]` pour tester le dernier élément d'un tableau.

En cas de correspondance, ce projecteur est allumé.

`repeatOrder` se raccourcit à mesure que des éléments sont supprimés du devant du tableau pour être comparés. Lorsque `repeatOrder.length` atteint zéro, la fonction `nextTurn` est appelée et la séquence est ajoutée et rejouée une nouvelle fois.

Si le joueur a choisi le mauvais projecteur, le message texte est modifié afin d'indiquer que la partie est terminée et `gameMode` est changé de manière qu'aucun autre clic de souris ne soit accepté :

```
// Réception des clics de souris sur les projecteurs
public function clickLight(event:MouseEvent) {
    // Empêcher les clics pendant l'affichage de la séquence
    if (gameMode != "replay") return;

    // Éteindre projecteur s'il ne s'est pas éteint lui-même
    lightOff(null);
```

```
        // Réponse exacte
        if (event.currentTarget.lightNum == repeatOrder.shift()) {
            lightOn(event.currentTarget.lightNum);

            // Vérifier si la séquence est terminée
            if (repeatOrder.length == 0) {
                nextTurn();
            }

        // Mauvaise réponse
        } else {
            textMessage.text = "Game Over!";
            gameMode = "gameover";
        }
    }
```

La valeur `"gameover"` de `gameMode` n'est en fait utilisée par aucun fragment de code. Puisqu'elle est différente de `"repeat"`, les clics ne seront cependant pas acceptés par les projecteurs, ce qui est exactement l'effet recherché.

Il ne reste plus maintenant que les accolades de fermeture des structures de classe et de paquetage. Elles viennent à la fin de chaque fichier de paquetage AS. Le jeu ne peut se compiler en leur absence.

Modifier le jeu

Au Chapitre 3, nous avons commencé par un jeu qui s'exécutait dans le scénario principal, un peu comme ici. À la fin du chapitre, nous avons cependant placé le jeu à l'intérieur d'un clip attitré (en réservant le scénario principal aux écrans d'introduction et de fin de partie).

Vous pouvez procéder de la même manière ici. Sinon vous pouvez renommer la fonction `MemoryGame` en l'appelant `startGame`. Elle ne sera ainsi plus déclenchée dès le départ de l'animation.

Vous pourriez placer un écran d'introduction dans la première image de l'animation, avec une commande stop dans l'image et un bouton pour émettre la commande `play` afin que l'animation se poursuive et passe à l'image suivante. Dans cette image, vous pourriez appeler la fonction `startGame` afin de lancer la partie.

Si vous souhaitez étendre ce jeu au-delà d'une image, vous devez changer `extends Sprite` au début de la classe en le remplaçant par `extends MovieClip`. Un sprite est un clip doté d'une seule image, tandis qu'un clip peut contenir plusieurs images.

Ensuite, au lieu d'afficher simplement le message "Game Over" lorsque le joueur rate un coup, vous pourriez supprimer tous les projecteurs et le message texte avec `removeChild` et passer directement à une nouvelle image.

Ces deux méthodes (encapsuler le jeu dans un clip ou attendre l'image 2 pour démarrer le jeu) permettent de créer une application plus complète.

L'une des modifications qui pourraient être apportées à ce jeu consisterait à commencer d'emblée avec une séquence de plusieurs étapes. Vous pourriez inclure au départ deux nombres aléatoires dans `playOrder` ; le jeu commencerait alors avec un total de trois éléments dans la séquence.

Une autre modification intéressante pourrait consister à faciliter le jeu en n'ajoutant dans la séquence que de nouveaux éléments qui ne correspondent pas au dernier. Par exemple, si le premier élément est 3, le suivant pourrait être 1, 2, 4 ou 5. Le fait de ne pas répéter les éléments réduit légèrement la complexité du jeu.

Vous pourriez pour cela utiliser une simple boucle `while` :

```
do {
    var r:uint = Math.floor(Math.random()*numLights);
} while (r == playOrder[playOrder.length-1]);
```

Vous pourriez également augmenter la vitesse à laquelle la séquence est rejouée. Pour l'instant, les projecteurs s'allument toutes les 1 000 millisecondes. Ils s'éteignent après 500 millisecondes. Vous pourriez ainsi stocker la valeur 1 000 dans une variable (comme `lightDelay`) puis réduire cette valeur de 20 millisecondes à chaque tour. Utilisez sa valeur entière pour `lightTimer` et la moitié de sa valeur pour `offTimer`.

Les variantes les plus intéressantes de ce jeu s'opèrent sans doute en apportant des modifications non pas au code mais plutôt aux graphismes. Pourquoi les projecteurs devraient-ils nécessairement être alignés ? Pourquoi leur donner à tous la même apparence ? Pourquoi d'ailleurs utiliser des projecteurs ?

Imaginez un jeu où les projecteurs seraient remplacés par des oiseaux chanteurs, tous différents et tous cachés dans une scène sylvestre. À mesure qu'ils ouvriraient leurs becs et pépieraient, vous seriez invité à vous rappeler non seulement lesquels ont sifflé mais également l'endroit où ils étaient situés.

Jeu de déduction

Voici un autre jeu classique. Comme le jeu de Memory, le jeu de déduction peut être joué avec un simple jeu de pièces. Il peut même être joué avec un crayon et du papier. Sans ordinateur, on ne peut cependant jouer qu'à deux joueurs. L'un doit produire une séquence en quelque manière aléatoire de couleurs tandis que l'autre doit s'efforcer de la deviner.

Codes sources

http://flashgameu.com

A3GPU04_Deduction.zip

Le jeu de déduction a été popularisé sous le nom de marque Mastermind. Il s'agit de l'un des jeux de découverte de code les plus simples.

Le jeu est généralement joué avec une séquence aléatoire de cinq pions, chacun d'une couleur parmi cinq possibles (par exemple rouge, vert, violet, jaune et bleu). Le joueur doit tenter une proposition pour chacun des cinq pions, même s'il peut ne proposer aucun choix pour un ou plusieurs pions. Le joueur peut donc par exemple proposer comme essai rouge, rouge, bleu, bleu, vide.

Lorsque le joueur fournit sa proposition, l'ordinateur retourne le nombre de pions correctement placés et le nombre de pions dont la couleur correspond à celle d'un pion requis mais qui ne se trouve pas au bon endroit pour le moment. Ainsi, si la séquence est rouge, vert, bleu, jaune, bleu et que le joueur propose la séquence rouge, rouge, bleu, bleu, vide, le résultat correspond à une couleur au bon emplacement et à une couleur exacte. Le joueur doit alors exploiter ces deux éléments d'information pour élaborer sa proposition suivante. Un bon joueur parvient généralement à deviner la séquence complète en moins de dix tentatives.

Mathématiquement, il est possible de deviner n'importe quelle séquence aléatoire en cinq tentatives uniquement. Il faut pour cela des algorithmes plutôt évolués. Pour plus d'informations à ce sujet, consultez l'article Wikipédia Mastermind en version anglaise.

Configurer l'animation

Nous allons configurer ce jeu de manière un peu plus robuste que le jeu de Simon. Il possédera trois images : une image d'introduction, une image de jeu et une image de fin de partie. Toutes les trois posséderont une apparence minimaliste afin que nous puissions nous concentrer sur le code ActionScript.

Un arrière-plan et un titre seront inclus dans les trois images. Vous pouvez le voir à la Figure 4.4.

L'image 1 contient un unique bouton. J'ai créé un objet d'affichage de bouton `BasicButton` simple dans la bibliothèque. Il ne contient en fait aucun texte : le texte qui apparaît à la Figure 4.4 est en réalité placé au-dessus du bouton dans l'image.

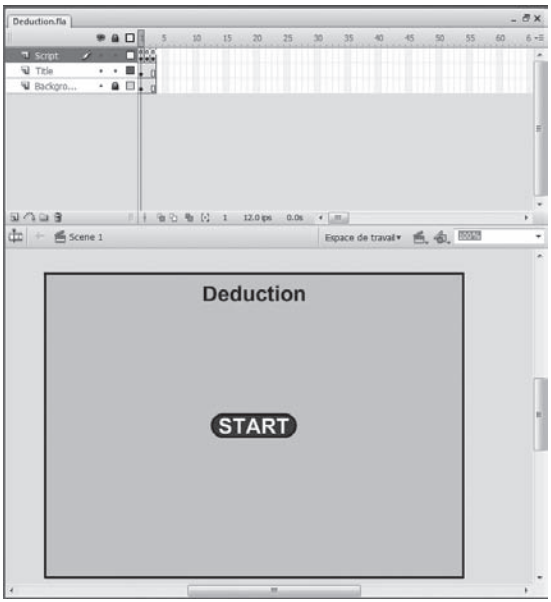


Figure 4.4
L'image 1 inclut un arrière-plan et un titre qui sont présents dans toutes les images, ainsi qu'un bouton Start qui ne figure que dans cette première image.

Le script de l'image 1 arrête l'animation à cette image et configure le bouton de manière à accepter un clic de souris pour démarrer le jeu :

```
stop();
startButton.addEventListener(MouseEvent.CLICK,clickStart);
function clickStart(event:MouseEvent) {
    gotoAndStop("play");
}
```

La seconde image, intitulée `"play"` dans le scénario, ne contiendra qu'une seule commande. Il s'agit d'un appel dans notre classe d'animation à une fonction nommée `startGame` que nous allons créer :

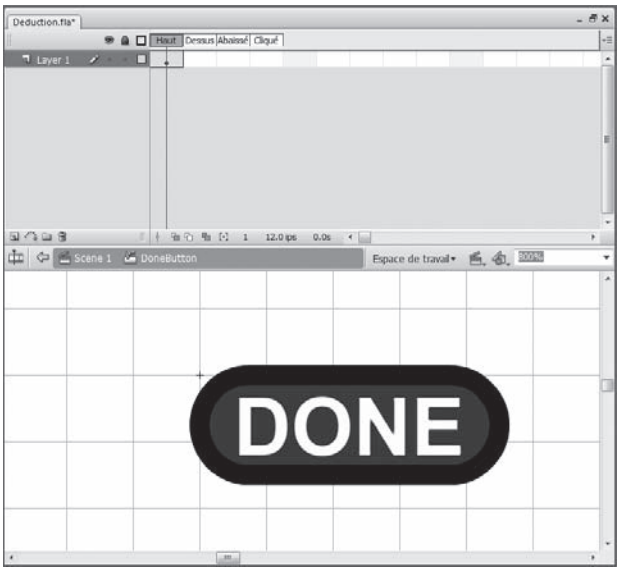
```
startGame();
```

La dernière image est intitulée `"gameover"` et possède sa propre copie du bouton de l'image 1. Le texte situé au-dessus est cette fois `"Play Again"`. Le script dans l'image est très similaire :

```
playAgainButton.addEventListener(MouseEvent.CLICK,clickPlayAgain);
function clickPlayAgain(event:MouseEvent) {
    gotoAndStop("play");
}
```

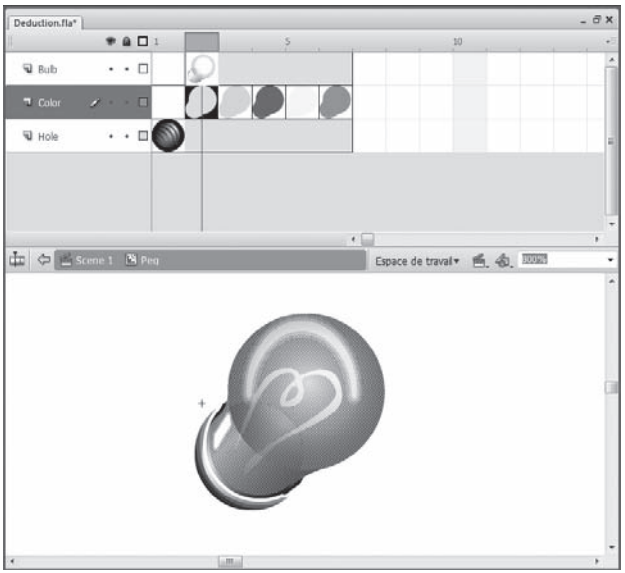
Il nous faut deux symboles en plus du symbole de bibliothèque `BasicButton`. Le premier est un petit bouton nommé `DoneButton`. Comme le montre la Figure 4.5, il inclut cette fois le texte qui l'illustre.

Figure 4.5
Le bouton Done utilisé dans le jeu fait la même hauteur que les trous à pion utilisés dans le jeu.



Le clip principal requis pour le jeu est le clip `Peg`. Il s'agit de plus que d'un simple pion. C'est une série de six images, la première présentant un trou vide et les cinq suivantes affichant cinq ampoules de couleur différente dans le trou. La Figure 4.6 présente ce clip.

Figure 4.6
Le clip Peg contient une prise d'ampoule vide et cinq images où cette prise est remplie par une ampoule de couleur différente.



À part l'arrière-plan et le titre, le scénario principal ne contient rien. Nous allons utiliser le code ActionScript pour créer tous les éléments du jeu. Et, cette fois, nous tiendrons le registre de chaque objet créé afin de pouvoir les supprimer en fin de partie.

Définition de la classe

L'animation de cet exemple est nommée **Deduction.fla** et le fichier ActionScript, **Deduction.as**. La classe du document dans l'inspecteur des propriétés doit donc être Deduction afin que l'animation utilise le fichier AS correspondant.

La définition de classe de ce jeu est un peu plus simple que celle du jeu de Simon. En effet, nous n'utiliserons cette fois ni minuteur ni son. Seules les classes `flash.display`, `flash.events` et `flash.text` seront donc importées : la première pour afficher et contrôler les clips, la deuxième pour réagir aux clics de souris et la troisième pour créer des champs texte :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
```

Pour la déclaration de classe, nous la ferons étendre `MovieClip` au lieu de `Sprite`. En effet, le jeu s'étendra sur trois images et non une. Or un sprite n'utilise qu'une image :

```
    public class Deduction extends MovieClip {
```

Pour ce jeu, nous utiliserons une large gamme de constantes. D'abord, nous définirons les constantes `numPegs` et `numColors`. Nous pourrons ainsi aisément modifier le jeu afin d'inclure plus de cinq pions dans la séquence ou plus ou moins d'options de couleur.

Nous incluerons également un ensemble de constantes afin de définir l'emplacement où seront dessinées les lignes de pions. Nous utiliserons un décalage horizontal et vertical pour toutes les lignes ainsi que l'espacement entre les lignes et l'espacement entre les pions. Nous facilitons ainsi l'ajustement de l'emplacement des pions en fonction de la taille des pions et des éléments environnants du jeu :

```
        // Constantes
        static const numPegs:uint = 5;
        static const numColors:uint = 5;
        static const maxTries:uint = 10;
        static const horizOffset:Number = 30;
        static const vertOffset:Number = 60;
        static const pegSpacing:Number = 30;
        static const rowSpacing:Number = 30;
```

Il nous faut deux variables principales pour tenir le registre de l'état d'avancement dans le jeu. La première est un tableau contenant la solution. Il s'agira d'un simple tableau de cinq nombres (par exemple 1, 4, 3, 1, 2). La seconde variable est `turnNum`, qui tient le registre du nombre de tentatives réalisées par le joueur :

```
// Variables de jeu
private var solution:Array;
private var turnNum:uint;
```

Dans ce jeu, nous allons tenir un registre précis de tous les objets d'affichage que nous créons. Il y aura une ligne courante de cinq pions stockée dans `currentRow`. Le texte à droite de chaque ligne de pions sera `currentText`. Le bouton à droite des pions sera `currentButton`. En outre, nous utiliserons le tableau `allDisplayObjects` pour tenir le registre de tout ce que nous créons :

```
// Références aux objets d'affichage
private var currentRow:Array;
private var currentText:TextField;
private var currentButton:DoneButton;
private var allDisplayObjects:Array;
```

Chaque classe a sa fonction constructeur, qui possède le même nom que la classe. Ici, nous ne l'utiliserons cependant pas, car le jeu ne commence pas dans la première image. Nous attendons que le joueur clique d'abord sur le bouton Start. Nous incluerons donc cette fonction, mais sans y insérer de code :

```
public function Deduction() {
}
```

Libre à vous de voir si vous souhaitez inclure une fonction constructeur vide. Pour ma part, il m'arrive très souvent de réaliser après coup que je dois réaliser quelque chose dans la fonction constructeur avant de finaliser mon jeu ; dans le doute, je l'ajoute donc d'emblée lorsque je commence une classe.

Démarrer une nouvelle partie

Lorsqu'une nouvelle partie commence, le scénario principal appelle `startGame` afin de créer la séquence de cinq pions que recherche le joueur. Cette fonction crée le tableau de solution et y insère cinq nombres aléatoires compris entre 1 et 5.

La variable `turnNum` est positionnée à 0. Ensuite, la fonction `createPegRow` est appelée pour faire le gros du travail :

```
// Créer la solution et afficher la première ligne de pions
public function startGame() {
    allDisplayObjects = new Array();
    solution = new Array();
    for(var i:uint=0;i<numPegs;i++) {
        // Aléatoire, compris entre 1 et 5
        var r:uint = uint(Math.floor(Math.random()*numColors)+1);
        solution.push(r);
    }
    turnNum = 0;
    createPegRow();
}
```

Créer une ligne de pions

C'est la fonction `createPegRow` qui réalise tout le travail qui consiste à créer les cinq pions ainsi que le bouton et le texte à côté. Nous l'appellerons à chaque fois qu'un tour commence.

Elle crée d'abord cinq nouvelles copies de l'objet `Peg` de la bibliothèque. Chaque objet est placé à l'écran en fonction des valeurs des constantes `pegSpacing`, `rowSpacing`, `horizOffset` et `vertOffset`. Chaque objet est également positionné à l'image 1, qui correspond au trou vide.

La commande `addEventListener` fait réagir chaque pion à un clic de souris. Nous activons aussi la propriété `buttonMode` des pions afin que le curseur change quand l'utilisateur les survole.

La propriété `pegNum` est ajoutée à chaque pion lorsqu'il est créé, afin de pouvoir identifier le pion sur lequel le joueur a cliqué.

Une fois que le pion est ajouté à l'écran avec `addChild`, il est également ajouté à `allDisplayObjects`. Ensuite, il est ajouté à `currentRow`, mais pas lui-même. Au lieu de cela, il est ajouté à `currentRow` sous la forme d'un petit objet avec les propriétés `peg` et `color`. La première est une référence au clip. La seconde est un nombre définissant la couleur, ou l'absence, du pion dans le trou :

```
// Créer une ligne de pions, plus le bouton DONE et le champ texte
public function createPegRow() {

    // Créer des pions et en faire des boutons
    currentRow = new Array();
    for(var i:uint=0;i<numPegs;i++) {
        var newPeg:Peg = new Peg();
        newPeg.x = i*pegSpacing+horizOffset;
```

```
        newPeg.y = turnNum*rowSpacing+vertOffset;
        newPeg.gotoAndStop(1);
        newPeg.addEventListener(MouseEvent.CLICK,clickPeg);
        newPeg.buttonMode = true;
        newPeg.pegNum = i;
        addChild(newPeg);
        allDisplayObjects.push(newPeg);

        // Enregistrer les pions sous forme de tableau d'objets
        currentRow.push({peg: newPeg, color: 0});
    }
```

Une fois que les cinq pions ont été créés, une copie du clip `DoneButton` est ajoutée à droite. Mais, tout d'abord, une vérification est opérée afin de voir si le `currentButton` existe déjà. Il n'existe pas la première fois qu'une ligne de pions est créée, aussi, le bouton est créé et ajouté à la scène. Il obtient aussi un écouteur événementiel et est ajouté à `allDisplayObjects`.

L'emplacement horizontal du bouton est déterminé par des constantes. Il doit se trouver à un `pegSpacing` de plus sur la droite du dernier pion dans la ligne. Nous n'avons besoin de définir que l'abscisse `x` du bouton lorsqu'il est créé car nous le déplaçons simplement vers le bas de l'écran avec chaque nouvelle ligne de pions ajoutée. La position `x` n'est définie que cette fois, mais la position `y` l'est à chaque fois que la fonction `createPegRow` est appelée :

```
// Ne créer le bouton DONE que si nous ne l'avons pas déjà fait
if (currentButton == null) {
    currentButton = new DoneButton();
    currentButton.x = numPegs*pegSpacing+horizOffset+pegSpacing;
    currentButton.addEventListener(MouseEvent.CLICK,clickDone);
    addChild(currentButton);
    allDisplayObjects.push(currentButton);
}
// Positionner le bouton DONE avec la ligne
currentButton.y = turnNum*rowSpacing+vertOffset;
```

Ajouter le champ texte

Après le bouton vient le champ texte. Celui-ci est décalé à droite d'un `pegSpacing` et de la largeur du `currentButton`. Nous ne réalisons aucune mise en forme particulière, afin de faire simple.

À la différence du bouton, un nouveau champ texte sera ajouté à chaque fois que nous créons une ligne de pions. Pour résoudre le puzzle, les joueurs doivent pouvoir examiner toutes leurs précédentes tentatives et les résultats obtenus.

Le `currentButton` est défini en tant que `DoneButton` au début de la classe. Il ne se voit cependant pas attribuer de valeur. Lorsque cette fonction s'apprête la première fois à l'utiliser, sa valeur est donc `null`. La plupart des objets sont positionnés à `null` au moment où ils sont créés. Les nombres sont cependant positionnés à zéro et ne peuvent jamais l'être à `null`.

Le champ texte commence par les instructions "Click on the holes to place pegs and click DONE" (cliquez sur les trous pour placer les pions et cliquez sur DONE). Il contiendra plus tard les résultats de chaque tentative à la fin des tours :

```
        // Créer le message texte à côté des pions et du bouton
        currentText = new TextField();
        currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
        currentText.y = turnNum*rowSpacing+vertOffset;
        currentText.width = 300;
        currentText.text = "Click on the holes to place pegs and click DONE.";
        addChild(currentText);
        allDisplayObjects.push(currentText);
    }
```

Examinez la Figure 4.7 afin de voir à quoi ressemble l'écran lorsque la première `createPegRow` est appelée. Vous verrez les cinq pions, suivis par le bouton, puis le champ texte.

Figure 4.7
Lorsque le jeu commence, la première ligne de trous de pion est créée et le joueur doit opérer sa première conjecture.

Vérification des propositions du joueur

Lorsque le joueur clique sur un trou de pion, il fait défiler en boucle les pions colorés et revient à un trou vide s'il clique assez de fois.

Pour savoir sur quel pion (`Peg`) le joueur a cliqué, nous examinerons `event.currentTarget.pegNum`. Nous aurons ainsi un index à rechercher dans le tableau `currentRow`. De là, nous pourrons obtenir les propriétés `color` et `Peg`.

Les couleurs sont numérotées de un à cinq, le zéro représentant l'absence de pion. Les images dans les clips sont cependant numérotées en commençant à 1. L'image 1 est le trou vide et les images 2 à 6 correspondent aux couleurs. Gardez-le à l'esprit lorsque vous examinez le code et l'ajout `+1` dans l'instruction `gotoAndStop`.

La couleur du `Peg` est stockée dans `currentColor`, une variable temporaire qui n'est utilisée que dans cette fonction. Si c'est inférieur au nombre de couleurs disponibles, le `Peg` affiche simplement la couleur suivante. Si la dernière couleur est déjà affichée, le `Peg` boucle pour afficher le trou vide, qui est la première image de la page :

```
// Le joueur clique sur un pion
public function clickPeg(event:MouseEvent) {
    // Déterminer le pion et récupérer la couleur
    var thisPeg:Object = currentRow[event.currentTarget.pegNum];
    var currentColor:uint = thisPeg.color;

    // Avancer couleur du pion d'une unité, boucler de 5 à 0
    if (currentColor < numColors) {
        thisPeg.color = currentColor+1
    } else {
        thisPeg.color = 0;
    }

    // Afficher le pion ou l'absence de pion
    thisPeg.peg.gotoAndStop(thisPeg.color+1);
}
```

Pour jouer, l'utilisateur déplace vraisemblablement le curseur sur chacun des cinq trous dans la ligne courante et clique le nombre requis de fois pour afficher la couleur de pion souhaitée. Après avoir fait cela pour les cinq trous, le joueur poursuit en cliquant sur le bouton Done.

Évaluer les déplacements du joueur

Lorsque le joueur clique sur le bouton Done, la fonction `clickDone` est appelée. Elle délègue ensuite l'exécution à la fonction `calculateProgress` :

```
// Le joueur clique sur le bouton DONE
public function clickDone(event:MouseEvent) {
    calculateProgress();
}
```

La fonction `calculateProgress` s'occupe de calculer dans quelle mesure la proposition du joueur correspond à la solution.

Les deux variables locales `numCorrectSpot` et `numCorrectColor` sont les principaux résultats que nous cherchons à calculer. Il est en fait très facile de déterminer `numCorrectSpot` : nous parcourons en boucle chacun des `Peg` et déterminons si la couleur sélectionnée par le joueur correspond à la solution. Si c'est le cas, nous ajoutons 1 à `numCorrectSpot`.

Le calcul de `numCorrectColor` est déjà plus complexe. Pour commencer, il faut ignorer tous les pions que l'utilisateur a correctement placés pour se concentrer sur les pions incorrects, puis examiner les couleurs que le joueur a sélectionnées et qui auraient pu aller autre part.

Une solution astucieuse pour cela consiste à tenir le registre de chacune des couleurs utilisées par le joueur dans ces pions incorrects. En outre, tenez le registre des couleurs requises pour les pions incorrects. Nous ferons cela avec les tableaux `solutionColorList` et `currentColorList`.

Chacun des éléments dans ces tableaux correspondra à une somme de chacune des couleurs trouvées. Par exemple, si deux rouges (couleur 0), un vert (couleur 1) et un bleu (couleur 2) sont trouvés, le tableau résultant sera `[2,1,1,0,0]. 2+1+1=4`. Puisqu'il y a cinq pions et que la somme fait quatre, nous devons avoir un pion correct et seuls quatre trous doivent être résolus.

Ainsi, si `[2,1,1,0,0]` représente les couleurs utilisées par le joueur dans les pions erronés et `[1,0,1,2,0]` représente les couleurs requises dans ces emplacements, nous pouvons déterminer le nombre de couleurs correctes utilisées aux mauvais emplacements en récupérant simplement le nombre le plus petit des deux tableaux : `[1,0,1,0,0]`.

Voyons cela couleur par couleur. Dans le premier tableau (`[2,1,1,0,0]`), le joueur place deux rouges. Mais le second tableau (`[1,0,1,2,0]`) montre que seul un rouge est requis. Le nombre minimal entre deux et un est un. Seul un rouge est hors de place. Le joueur a également placé un vert. Mais le second tableau montre qu'aucun vert n'est requis. Le plus petit nombre est donc zéro. Le joueur a sélectionné un bleu et un bleu est requis. En voilà un autre dans le tableau. Le joueur a choisi zéro jaune, mais deux sont requis. Le joueur a choisi zéro violet et zéro est requis. Voilà un autre zéro dans le tableau. Le plus petit est zéro. Ainsi 1+0+1+0+0 = 2. Deux couleurs sont mal placées. Le Tableau 4.2 présente un autre aperçu de ce calcul.

Tableau 4.2 : Calcul des pions mal placés

Couleur	*Choisie par l'utilisateur*	*Correcte*	*Nombre de pions mal positionnés*
Rouge	2	1	1
Vert	1	0	0
Bleu	1	1	1
Jaune	0	2	0
Violet	0	0	0
Total mal positionnés			2

En plus de calculer le nombre total de pions corrects et de couleurs mal placées, nous allons en profiter pour désactiver le mode bouton des pions en utilisant `removeEventListener` et en positionnant `buttonMode` à `false` :

```
// Calcul du résultat
public function calculateProgress() {
    var numCorrectSpot:uint = 0;
    var numCorrectColor:uint = 0;
    var solutionColorList:Array = new Array(0,0,0,0,0);
    var currentColorList:Array = new Array(0,0,0,0,0);

    // Parcours en boucle des pions
    for(var i:uint=0;i<numPegs;i++) {
        // Ce pion est-il correct ?
        if (currentRow[i].color == solution[i]) {
            numCorrectSpot++;
        } else {
            // Pas de correspondance, mais enregistrer les couleurs pour le test suivant
            solutionColorList[solution[i]-1]++;
            currentColorList[currentRow[i].color-1]++;
        }
        // Désactiver le mode bouton du pion
        currentRow[i].peg.removeEventListener(MouseEvent.CLICK,clickPeg);
        currentRow[i].peg.buttonMode = false;
    }

    // Obtenir le nombre correct de couleurs correctement positionnées
    for(i=0;i<numColors;i++) {
        numCorrectColor += Math.min(solutionColorList[i],currentColorList[i]);
    }
```

Maintenant que nous connaissons le résultat des tests, nous pouvons les afficher dans le champ texte qui contenait précédemment les instructions :

```
// Afficher le résultat
currentText.text = "Correct Spot: "+numCorrectSpot+", Correct Color: "+numCorrectColor;
```

Ensuite, nous souhaitons passer au `turnNum` et vérifier si le joueur a trouvé la solution. Si c'est le cas, nous repassons le contrôle à la fonction `gameOver`. Par ailleurs, si le joueur a excédé le nombre maximal de tentatives, nous passons le contrôle à la fonction `gameLost`.

Si la partie n'est pas terminée, nous passons au tour suivant en appelant `createPegRow` :

```
        turnNum++;

        if (numCorrectSpot == numPegs) {
            gameOver();
        } else {
            if (turnNum == maxTries) {
                gameLost();
            } else {
                createPegRow();
            }
        }
    }
```

Fin de la partie

Si le joueur a trouvé la solution, nous souhaitons lui indiquer que la partie est terminée. Nous poursuivrons en créant une ligne de plus dans le jeu. Cette fois, il n'est pas nécessaire d'afficher les pions car la ligne précédente que le joueur a complétée contient maintenant la solution correspondante. La Figure 4.8 montre à quoi ressemble l'écran.

Indiquer que le joueur a gagné

La nouvelle ligne n'a besoin de contenir que le bouton et un nouveau champ texte. Le bouton sera relié de manière à déclencher la fonction `clearGame`. Le champ texte suivant affichera "You Got It!" (trouvé). Nous devons l'ajouter à `allDisplayObjects` comme tout ce que nous créons dans le jeu :

```
// Le joueur a trouvé la solution
public function gameOver() {
    // Changer le bouton
    currentButton.y = turnNum*rowSpacing+vertOffset;
    currentButton.removeEventListener(MouseEvent.CLICK,clickDone);
    currentButton.addEventListener(MouseEvent.CLICK,clearGame);
```

```
        // Créer le message texte à côté des pions et du bouton
        currentText = new TextField();
        currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
        currentText.y = turnNum*rowSpacing+vertOffset;
        currentText.width = 300;
        currentText.text = "You got it!";
        addChild(currentText);
        allDisplayObjects.push(currentText);
    }
```

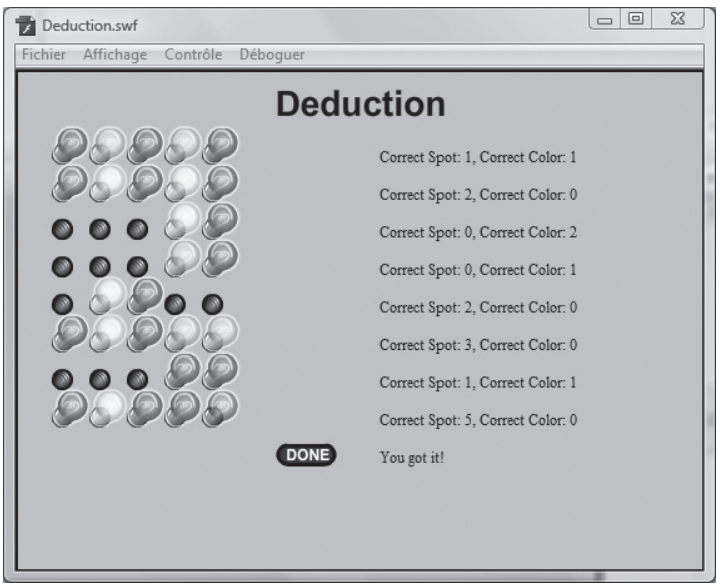


Figure 4.8
La partie se termine lorsque le joueur trouve la solution.

Indiquer que le joueur a perdu

La fonction `gameLost` est analogue à la fonction `gameOver`. Sa principale différence tient à ce qu'elle doit créer une dernière ligne de pions afin de révéler la solution au joueur qui donne sa langue au chat. La Figure 4.9 montre à quoi pourrait ressembler l'écran à ce point.

Ces nouveaux pions n'ont pas besoin d'être configurés comme des boutons. Ils doivent cependant être configurés pour afficher la bonne couleur de pion.

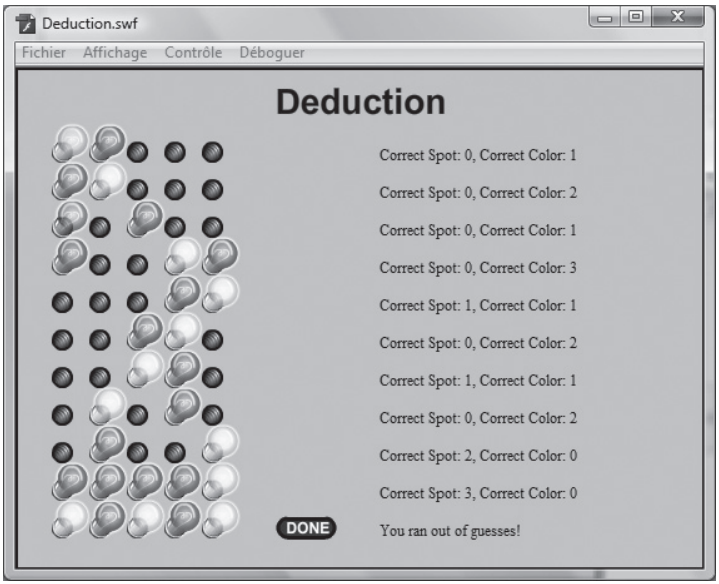


Figure 4.9
Le joueur a épuisé tous ses essais.

En outre, le bouton Done est modifié afin d'appeler la fonction `clearGame`, comme dans `gameOver`. Un nouveau champ texte est créé, mais qui affiche cette fois "You ran out of guesses" :

```
// Le joueur a atteint le nombre maximal de tours
public function gameLost() {
    // Changer le bouton
    currentButton.y = turnNum*rowSpacing+vertOffset;
    currentButton.removeEventListener(MouseEvent.CLICK,clickDone);
    currentButton.addEventListener(MouseEvent.CLICK,clearGame);

    // Créer le message texte à côté des pions et du bouton
    currentText = new TextField();
    currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
    currentText.y = turnNum*rowSpacing+vertOffset;
    currentText.width = 300;
    currentText.text = "You ran out of guesses!";
    addChild(currentText);
    allDisplayObjects.push(currentText);

    // Créer la ligne de pions finale pour afficher la réponse
    currentRow = new Array();
```

```
        for(var i:uint=0;i<numPegs;i++) {
            var newPeg:Peg = new Peg();
            newPeg.x = i*pegSpacing+horizOffset;
            newPeg.y = turnNum*rowSpacing+vertOffset;
            newPeg.gotoAndStop(solution[i]+1);
            addChild(newPeg);
            allDisplayObjects.push(newPeg);
        }

    }
```

Lorsque la partie se termine, le bouton Done reste à l'écran dans la ligne finale. S'il clique dessus, le joueur est conduit à l'écran de fin de partie du scénario principal, où il peut choisir de jouer à nouveau. Avant de faire cela, nous devons cependant effacer tous les éléments de jeu de la scène.

Effacer les éléments du jeu

La suppression des objets d'affichage est un processus à plusieurs étapes que facilite grandement notre tableau `allDisplayObjects`. Chacun des objets d'affichage que nous avons créés, qu'il s'agisse d'un clip, d'un bouton ou d'un champ texte, a été ajouté à ce tableau. Nous pouvons maintenant les supprimer en parcourant le tableau en boucle et en utilisant `removeChild` pour retirer l'objet de l'écran :

```
// Supprimer tout pour aller à l'écran de fin de partie
public function clearGame(event:MouseEvent) {
    // Supprimer tous les objets d'affichage
    for(var i in allDisplayObjects) {
        removeChild(allDisplayObjects[i]);
    }
```

Même à l'écart de la scène, les objets existent toujours, en l'attente d'une commande `addChild` qui les replacerait à l'écran. Pour s'en débarrasser véritablement, nous devons supprimer toutes les références à ces objets. Nous faisons référence aux objets d'affichage à plusieurs endroits de notre code, dont le `allDisplayObjects`. Si nous le positionnons à `null`, puis positionnons les variables `currentText`, `currentButton` et `currentRow` à `null` également, aucune des variables ne fait plus référence à aucun de nos objets d'affichage. Les objets sont ensuite supprimés.

En réalité, lorsque vous supprimez toutes les références à un objet d'affichage, vous le mettez à disposition du "ramasse-miettes". Cela signifie simplement que Flash peut supprimer ces objets de la mémoire à tout moment. Comme il n'y a plus de référence à ces objets et aucun moyen de faire référence à l'un d'entre eux si vous le souhaitez, vous pouvez pour votre part les considérer comme supprimés. Flash ne se compliquera toutefois pas la vie à les supprimer sur-le-champ ; il ne s'y attellera que lorsqu'il aura quelques cycles processeur à gaspiller pour cela.

```
        // Positionner toutes les références d'objets d'affichage à null
        allDisplayObjects = null;
        currentText = null;
        currentButton = null;
        currentRow = null;
```

Pour finir, la fonction `clearGame` demande au scénario principal de se rendre à l'image `gameover`. C'est à cet endroit que figure un bouton Play Again sur lequel le joueur peut cliquer. Tous les objets d'affichage étant supprimés, le jeu peut véritablement recommencer, avec de nouveaux objets d'affichage :

```
        // Demander au scénario principal d'avancer
        MovieClip(root).gotoAndStop("gameover");
    }
```

La fonction `clearGame` est la fonction critique utilisée pour créer un jeu dans le scénario principal qui peut être effacé et redémarré. Si on la compare au fonctionnement du jeu de Memory du Chapitre 3, le résultat semble identique. Vous avez la certitude que, la deuxième fois que le joueur commence le jeu, il entame une partie entièrement nouvelle, comme la première fois.

L'approche utilisée au Chapitre 3 est cependant un peu plus simple à mettre en place parce que tous les objets d'affichage sont instantanément et aisément supprimés lorsque le clip disparaît dans l'image de fin de partie.

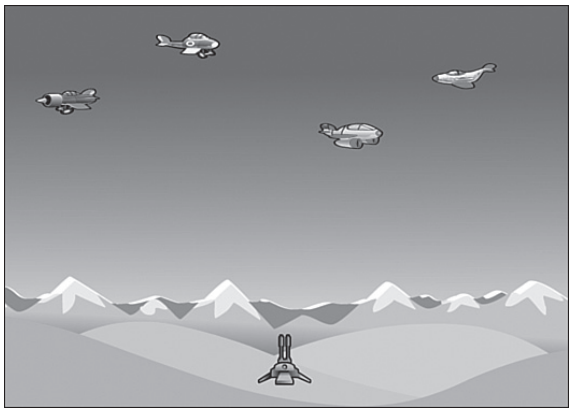
Modifier le jeu

Comme pour le jeu de Simon, l'une des meilleures variantes du jeu de déduction consiste à intervenir sur les graphismes du jeu. Vous pouvez utiliser toutes sortes d'objets pour les pions et même créer une mise en scène pour les situer dans un contexte particulier. Par exemple, vous pouvez proposer au joueur de tenter d'ouvrir un coffre-fort ou de déverrouiller une porte dans un jeu d'aventure.

Notre utilisation des constantes permet aisément de proposer un nombre de tentatives plus élevé dans ce jeu, de réduire ou d'augmenter le nombre de pions ou de couleurs, etc. Si vous souhaitez proposer un plus grand nombre d'essais, vous aurez sans doute besoin d'agrandir la scène ou de réduire l'espacement des lignes.

Pour finaliser ce jeu, j'utiliserais d'abord un objet `TextFormat` afin de mettre en forme le message texte. Ensuite, j'ajouterais des instructions dans l'écran d'introduction. Un bouton Restart (redémarre) dans l'image du jeu pourrait aussi permettre au joueur de recommencer à tout moment. Il pourrait tout simplement appeler `clearGame` afin de supprimer tous les éléments écran et conduire à l'image de fin de partie.

Pour rendre ce jeu plus proche du jeu de société Mastermind, vous pourriez remplacer le message texte par des pions blancs et noirs. L'un désignerait un pion correctement placé, l'autre, une couleur correcte située au mauvais emplacement. Noir, noir, blanc signifierait ainsi deux pions corrects et une couleur correcte au mauvais emplacement.



5

Animation de jeu : jeux de tir et de rebond

Au sommaire de ce chapitre :

- Animation de jeu
- Air Raid
- Casse-brique

Jusqu'à présent, nous n'avons créé que des jeux dont les éléments restaient positionnés à un même emplacement. Ces éléments changaient et pouvaient être modifiés par les actions de l'utilisateur, mais ils ne se déplaçaient pas.

Dans ce chapitre, nous allons travailler avec des éléments de jeu animés. Certains seront contrôlés par le joueur, d'autres se déplaceront de manière autonome.

Après avoir examiné certains exemples d'animation, nous créerons deux jeux. Le premier s'appelle Air Raid, un jeu simple dans lequel vous contrôlez un canon de DCA et tentez de toucher les avions qui traversent le ciel au-dessus de votre tête. Le second est un jeu de casse-brique où vous contrôlez une raquette et renvoyez une balle en direction d'un mur de briques qui disparaissent quand la balle les touche.

Animation de jeu

Codes sources

http://flashgameu.com

A3GPU05_Animation.zip

Au Chapitre 2, nous avons examiné deux types d'animations principaux : les animations à images et les animations temporelles. Nous n'utiliserons dans ce chapitre que des animations temporelles, parce qu'elles sont plus fiables et offrent des résultats de meilleure apparence.

Animation temporelle

L'idée de base de l'animation temporelle est de déplacer les objets à cadence homogène, quelles que soient les performances du lecteur Flash.

Un unique mouvement, que nous appellerons une étape, se produit à chaque image. Une cadence d'images de 12 ips implique donc 12 étapes par seconde. Bien que les étapes d'animation se produisent à chaque image, il ne s'agit pas d'une animation d'image car nous déterminons la taille de chaque étape de manière temporelle.

À chaque étape, nous calculons le temps écoulé depuis l'étape précédente. Nous déplaçons les éléments du jeu en fonction de cette différence temporelle.

Si la première étape prend 84 millisecondes et la seconde, 90, nous déplaçons les objets légèrement plus loin à la seconde étape qu'à la première.

La Figure 5.1 présente un diagramme de trois images du mouvement avec une animation temporelle.

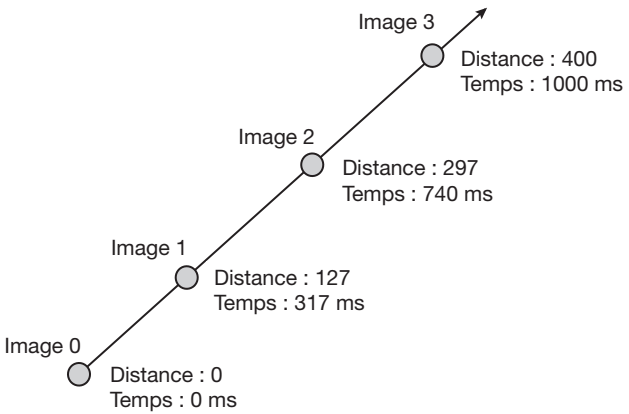


Figure 5.1
L'objet avance de 400 pixels par seconde quelle que soit la cadence d'images.

L'objet à la Figure 5.1 est supposé se déplacer de 400 pixels à chaque seconde. L'animation est configurée pour une cadence d'images très faible de 4 ips. Pour compliquer les choses, supposons que l'ordinateur manque totalement de réactivité, par exemple parce qu'il est occupé par d'autres applications ou une activité réseau. Il n'est pas à même de proposer une cadence d'images constante ni même de générer quatre images par seconde.

Lors du développement d'animations temporelles, il est judicieux de modifier fréquemment la cadence d'images de votre animation pendant vos tests. En général, j'alterne entre 12 et 60 images par seconde. Mon but est d'obtenir un jeu qui possède une excellente apparence à 60 ips mais reste tout aussi jouable à 12 ips.

Si je relie accidentellement un élément de jeu à la cadence d'images au lieu de la mesure temporelle, je remarque ainsi rapidement une considérable différence dans le jeu entre les deux cadences d'images.

Lorsque la première image passe et qu'un événement `ENTER_FRAME` déclenche la fonction d'animation dans notre code, 317 millisecondes se sont écoulées. À 4 images par seconde, seules 250 millisecondes auraient dû s'écouler. Jusque-là, la cadence d'images est déjà à la traîne.

Mais, en utilisant la mesure temporelle de 317 millisecondes, nous pouvons calculer que l'objet aurait dû se déplacer d'une distance de 127 pixels. Cela fait 400 pixels par seconde multipliés par 0,317 seconde. À la cadence d'images, l'objet se trouve ainsi exactement où il doit être.

La seconde image prend plus de temps encore, pour un total de 423 millisecondes supplémentaires. Voilà en tout 740 millisecondes, ce qui place l'objet à 297 pixels de distance.

Ensuite, dans la dernière image de l'exemple, 260 millisecondes supplémentaires s'écoulent. Cela nous porte à un total exact de 1 000 millisecondes. La distance est donc de 400. Après 1 seconde, l'objet s'est donc déplacé de 400 pixels, en dépit du fait que l'animation a produit une cadence d'images irrégulière et n'est pas parvenue à générer 4 images par seconde.

Si nous avions simplement fait avancer l'objet de 100 pixels par image, il se serait maintenant déplacé de 300 pixels, n'ayant eu que 3 images pour avancer. Ce résultat pourrait être différent sur un autre ordinateur ou à un autre moment sur le même ordinateur, lorsque les performances seraient suffisantes pour produire 4 images par seconde.

Programmer des animations temporelles

L'astuce pour programmer des animations temporelles consiste à surveiller précisément le temps écoulé. En examinant la fonction `getTimer`, vous pouvez obtenir le nombre de millisecondes depuis que l'animation a commencé. La valeur brute de `getTimer` n'est pas intéressante en elle-même. Ce qui compte, c'est la différence temporelle entre les images.

Par exemple, il peut falloir 567 millisecondes pour que votre animation s'initialise et place les éléments à l'écran. Ainsi, la première image se produit à 567 et la seconde, à 629. La différence est de 62 millisecondes : c'est cette mesure qui nous permet de déterminer la distance qu'un objet doit avoir parcouru entre les images.

L'animation **AnimationTest.fla** contient un clip de cercle simple qui illustre le principe de l'animation temporelle. L'animation utilise **AnimationTest.as** comme script principal et **AnimatedObject.as** comme classe du clip.

La classe `AnimatedObject` possède une fonction constructeur qui accepte des paramètres. Cela signifie que, lorsque vous créez un nouveau `AnimatedObject`, vous devez passer des paramètres, comme ceci :

```
var myAnimatedObject:AnimatedObject = new AnimatedObject(100,150,5,-8);
```

Les quatre paramètres représentent l'emplacement horizontal et vertical, puis la vitesse horizontale et verticale du clip.

Voici la déclaration de classe, les déclarations de variable et la fonction `AnimatedObject`. Vous remarquerez les quatre paramètres, définis simplement sous les noms `x`, `y`, `dx`, `dy` :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class AnimatedObject extends MovieClip {
        private var speedX, speedY:Number; // Vitesse actuelle en pixels par seconde
        private var lastTime:int; // Mémoriser le temps de la dernière image
```

```
public function AnimatedObject(x,y,dx,dy) {
    // Définir emplacement et vitesse
    this.x = x;
    this.y = y;
    speedX = dx;
    speedY = dy;
    lastTime = getTimer();
    // Déplacer chaque image
    addEventListener(Event.ENTER_FRAME, moveObject);
}
```

L'usage de `dx` et de `dy` pour le stockage de la "différence sur l'axe x" et de la "différence sur l'axe y" est une pratique assez courante. Dans ce chapitre et les suivants, nous utiliserons ces deux noms de variable assez souvent.

La fonction prend quatre paramètres et les applique. Les deux premiers sont utilisés pour définir l'emplacement du clip. Les deux autres sont stockés dans `speedX` et `speedY`.

Ensuite, la variable `lastTime` est initialisée avec la valeur courante de `getTimer()`. Pour finir, `addEventListener` permettra à la fonction `moveObject` de s'exécuter à chaque image.

La fonction `moveObject` calcule d'abord le temps écoulé, puis l'ajoute à `lastTime`. La valeur de `timePassed` est ensuite utilisée pour calculer le déplacement.

En ajoutant `timePassed` à `lastTime`, vous vous assurez qu'aucun temps n'est perdu dans l'animation. Si au lieu de cela vous positionnez `lastTime` à `getTimer()` à chaque étape de l'animation, vous risquez de perdre de petites tranches de temps entre le moment où `timePassed` est calculé et celui où `lastTime` se voit attribuer sa valeur.

Comme `timePassed` s'exprime en millièmes de seconde (millisecondes), nous le divisons par 1 000 pour obtenir la quantité à multiplier par `speedX` et `speedY`. Par exemple, si `timePassed` vaut 100, cela équivaut à 100/1000 ou à 0,1 seconde. Si `speedX` vaut 23, l'objet se déplace de 23 × 0,1 ou de 2,3 pixels vers la droite :

```
// Déplacer en fonction de la vitesse
public function moveObject(event:Event) {
    // Calcul du temps écoulé
    var timePassed:int = getTimer() - lastTime;
    lastTime += timePassed;
```

```
            // Mettre à jour la position selon la vitesse et le temps
            this.x += speedX*timePassed/1000;
            this.y += speedY*timePassed/1000;
        }
    }
}
```

L'un des moyens simples de tester cette classe `AnimatedObject` consiste à utiliser une classe d'animation principale comme ceci :

```
package {
    import flash.display.*;
    public class AnimationTest extends MovieClip {

        public function AnimationTest() {
            var a:AnimatedObject = new AnimatedObject(100,150,5,-8);
            addChild(a);
        }
    }
}
```

Ce code crée un clip à 100, 150 qui se déplace à une vitesse de 5 horizontalement et de –8 verticalement. La classe `AnimatedObject` nous a donc en fait permis d'ajouter un objet mouvant à la scène à l'aide de deux simples lignes de code.

Un meilleur test de la classe `AnimatedObject` consiste à ajouter plusieurs objets et à les faire bouger au hasard dans toutes les directions. Voici une version de la classe principale qui se charge de cette tâche :

```
package {
    import flash.display.*;

   public class AnimationTest extends MovieClip {

        public function AnimationTest() {
            // Créer 50 objets positionnés au hasard avec des vitesses aléatoires
            for(var i:uint=0;i<50;i++) {
                var a:AnimatedObject =
new AnimatedObject(Math.random()*550, Math.random()*400, getRandomSpeed(), getRandomSpeed());
                addChild(a);
            }
        }
```

```
            // Obtenir une vitesse comprise entre 70 et 100, positive ou négative
            public function getRandomSpeed() {
                var speed:Number = Math.random()*70+30;
                if (Math.random() > .5) speed *= -1;
                return speed;
            }
        }
    }
```

Dans cette version de la classe, nous créons un nouvel `AnimatedObject` avec un emplacement et une vitesse aléatoires. La position aléatoire est créée en utilisant simplement `Math.random`. Pour la vitesse aléatoire, j'ai cependant utilisé une fonction séparée qui retourne une valeur positive ou négative comprise entre 70 et 100. Ce code sert à éviter que des objets se déplacent à une vitesse proche de 0.

La Figure 5.2 présente cette animation lorsqu'elle s'exécute la première fois. Les objets sont éparpillés à l'écran.

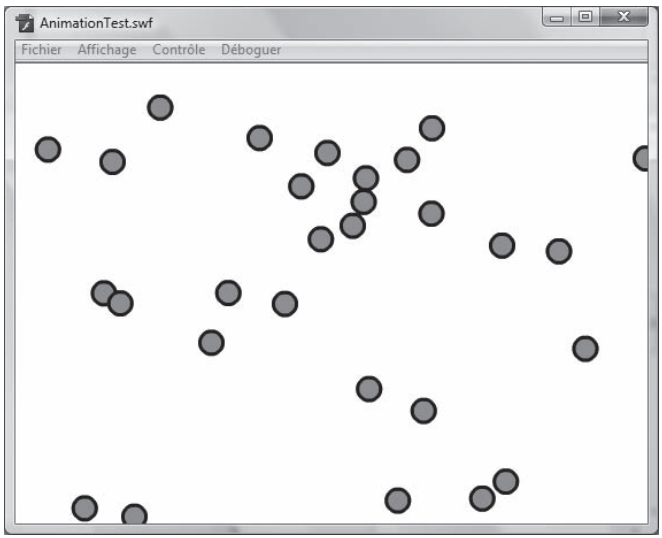


Figure 5.2
L'animation AnimationTest place cinquante objets aléatoires sur la scène.

Vous pouvez vous amuser un peu avec cette classe afin de créer des effets intéressants. Par exemple, si tous les objets commencent au même emplacement, vous obtiendrez un effet d'explosion.

Vous pouvez également ajuster le nombre d'objets créé et la cadence d'images de l'animation afin de voir comment votre ordinateur parvient à gérer une forte charge d'animation.

À présent, utilisons cette technique dans un jeu qui combine trois différents types d'objets animés.

Air Raid

Codes sources
http://flashgameu.com
A3GPU05_AirRaid.zip

Air Raid est analogue à certains anciens jeux d'arcade. La plupart d'entre eux s'inspiraient de scènes navales où le joueur commandait un sous-marin et tirait sur des bateaux qui striaient la surface de l'eau. Le plus ancien a probablement été Sea Wolf. Il proposait une vue subjective dans un périscope d'une scène où vous deviez tirer sur des cibles. Il s'agissait en fait d'une version en jeu vidéo des jeux électroniques Periscope, Sea Raider et Sea Devil.

Ces jeux de tir navals étaient probablement les plus simples à réaliser aux premières heures de la programmation informatique, car les bateaux et les torpilles ont l'intérêt de se déplacer plus lentement que les avions et les canons anti-DCA.

Dans notre jeu, le joueur déplace une tourelle anti-DCA en bas de l'écran à l'aide des touches fléchées du clavier. Il tire vers le haut en direction des avions qui défilent et tente d'en atteindre autant que possible avec un quantité de munitions limitée.

Configuration de l'animation et méthode

L'occasion est toute trouvée de créer maintenant un jeu qui utilise plusieurs classes. Nous avons pour l'essentiel trois différents types d'objets : des avions, une tourelle-canon et des balles. En créant une unique classe pour chacun de ces objets, nous pouvons créer le jeu étape par étape puis spécialiser le code pour chacun.

Il nous faut trois clips pour accompagner les trois classes. Les clips `AAGun` et `Bullet` feront une image chacun. Le clip `Airplane` en contiendra plusieurs, avec un dessin d'avion différent dans chacune. La Figure 5.3 présente ce clip. La sixième image à la fin contient un graphisme d'explosion que nous utiliserons lorsqu'un avion est touché.

En plus des trois fichiers de classe **AAGun.as**, **Airplane.as** et **Bullet.as**, il nous faut un fichier de classe principale pour l'animation, **AirRaid.as**.

Avions volants

La classe ActionScript pour les avions ne sera pas très différente quant à sa structure de la classe `AnimatedObject`, vue précédemment dans ce chapitre. Elle acceptera des paramètres dans la fonction constructeur pour déterminer la position de départ et la vitesse de l'avion. Elle utilisera

la mesure temporelle pour déterminer la différence entre les images. Elle utilisera un événement `ENTER_FRAME` pour faire avancer l'animation.

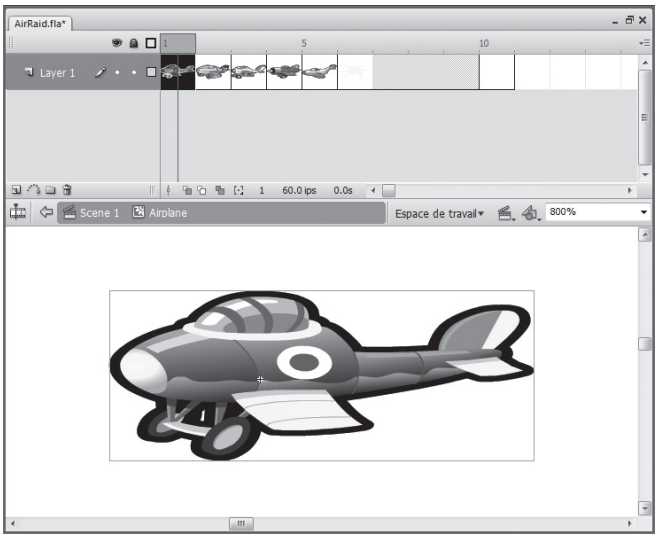


Figure 5.3
Le clip `Airplane` contient cinq avions différents situés chacun dans une image.

Déclaration de classe et de variables

Le code qui suit correspond à la définition de la classe et des variables que la classe utilisera. Comme l'avion ne volera qu'horizontalement, il n'aura besoin que de `dx`, la vitesse horizontale :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class Airplane extends MovieClip {
        private var dx:Number; // Vitesse et direction
        private var lastTime:int; // Temps d'animation
```

La fonction constructeur

La fonction constructeur prendra trois paramètres : `side` (côté), `speed` (vitesse) et `altitude`. Le paramètre `side` sera soit `"left"` (gauche) soit `"right"` (droite) selon le côté de l'écran d'où provient l'avion.

Le paramètre `speed` sera utilisé pour définir la variable `dx`. Si l'avion vient de la droite de l'écran, nous placerons automatiquement un signe moins devant la vitesse. Un avion progressant de gauche à droite à une vitesse de 80 aura donc une vitesse `dx` de `80`. À l'inverse, un avion progressant de droite à gauche avec une vitesse de 80 aura une vitesse `dx` de `-80`.

L'altitude n'est qu'un nom un peu pompeux pour désigner la position verticale de l'avion. 0 correspondra au haut de l'écran, 50 à 50 plus bas, etc.

En plus de définir l'emplacement et `dx`, nous aurons également besoin de renverser l'avion afin qu'il s'oriente dans la bonne direction. C'est ce que nous faisons en utilisant la propriété `scaleX` du clip. La valeur `-1` renverse l'image :

```
public function Airplane(side:String, speed:Number, altitude:Number) {
    if (side == "left") {
        this.x = -50; // Démarrer à gauche
        dx = speed; // Voler de gauche à droite
        this.scaleX = -1; // Renverser
    } else if (side == "right") {
        this.x = 600; // Démarrer à droite
        dx = -speed; // Voler de droite à gauche
        this.scaleX = 1; // Ne pas renverser
    }
    this.y = altitude; // Position verticale

    // Configurer l'animation
    addEventListener(Event.ENTER_FRAME,movePlane);
    lastTime = getTimer();
}
```

La fonction `Airplane` se termine en positionnant le minuteur événementiel et en initialisant la propriété `lastTime` comme nous l'avons fait dans la classe `AnimatedObject`.

Déplacer l'avion

La fonction `movePlane` calcule d'abord le temps écoulé, puis déplace l'avion en fonction du temps écoulé et de sa vitesse.

Ensuite, elle vérifie si l'avion a terminé son parcours de l'écran. Si c'est le cas, nous appelons la fonction `deletePlane` :

```
public function movePlane(event:Event) {
    // Calcul du temps écoulé
    var timePassed:int = getTimer()-lastTime;
    lastTime += timePassed;

    // Déplacer l'avion
    this.x += dx*timePassed/1000;

    // Vérifier s'il est sorti de l'écran
    if ((dx < 0) && (x < -50)) {
        deletePlane();
```

```
        } else if ((dx > 0) && (x > 600)) {
            deletePlane();
        }

    }
```

Supprimer les avions

La fonction `deletePlane` est une sorte de fonction autonettoyante, comme le montre le bloc de code suivant. Elle supprime l'avion de la scène avec une commande `removeChild`. Elle supprime ensuite l'écouteur de la fonction `movePlane`.

Il est toujours judicieux d'inclure une fonction avec une classe qui supprime l'objet. La classe peut ainsi gérer la suppression de ses propres écouteurs et de toutes les commandes requises pour nettoyer les autres références à elle-même.

Pour que l'avion disparaisse complètement, nous devons indiquer à la classe principale qu'il a terminé son parcours. Nous commençons donc ici par appeler `removePlane`, une fonction de la classe du scénario principal. C'est le scénario principal qui a créé l'avion au départ et qui, se faisant, le stocke dans un tableau. La fonction `removePlane`, à laquelle nous viendrons dans un instant, supprime l'avion du tableau :

```
// Supprimer l'avion de la scène et de la liste des avions
public function deletePlane() {
    MovieClip(parent).removePlane(this);
    parent.removeChild(this);
    removeEventListener(Event.ENTER_FRAME,movePlane);
}
```

Une fois que toutes les références à un objet ont été réinitialisées ou supprimées, le lecteur Flash réclamera la mémoire utilisée par l'objet.

Il y aura une seconde fonction pour supprimer l'avion. Celle-ci s'apparente à `deletePlane`, mais elle gérera le cas où l'avion est touché par le tir du joueur. Elle interrompra aussi l'événement d'image et indiquera à la classe principale de renvoyer l'avion du tableau. Au lieu de supprimer l'enfant de la scène, elle demandera cependant simplement au clip d'aller à l'image intitulée `"explode"` et de poursuivre la lecture à cet endroit.

Le clip contient un graphisme d'explosion qui démarre à l'image 6. Il se poursuit sur quelques images, puis atteint une image contenant une commande `parent.removeChild(this);` et une commande `stop();`. Ce code finalise la suppression de l'avion, après un bref aperçu de l'explosion pour le plaisir des yeux du joueur :

```
// Avion touché, afficher l'explosion
public function planeHit() {
    removeEventListener(Event.ENTER_FRAME,movePlane);
    MovieClip(parent).removePlane(this);
    gotoAndPlay("explode");
}
```

Vous pouvez faire durer l'explosion plus longtemps en rallongeant le nombre d'images entre l'image `"explosion"` et la dernière image contenant le script. De la même manière, vous pouvez placer une explosion animée dans ces images, sans qu'il soit nécessaire d'ajouter d'autre code ActionScript.

Tester la classe Airplane

C'est le scénario principal qui se charge de créer les avions, mais également de les supprimer. Nous créerons cette classe par la suite. Si nous souhaitons tester la classe `Airplane`, nous pouvons le faire avec une classe principale simple comme la suivante :

```
package {
    import flash.display.*;

    public class AirRaid extends MovieClip {
        public function AirRaid() {
            var a:Airplane = new Airplane("right",170,30);
            addChild(a);
        }
    }
}
```

Si vous effectuez un test, il est judicieux d'essayer différentes valeurs pour les paramètres. Par exemple, essayez `"left"` avec une vitesse de `30`. Testez autant de valeurs qu'il le faut pour vous assurer que la classe `Airplane` fonctionne avant de passer à la classe suivante.

Tourelle mouvante

La classe qui contrôle le canon anti-DCA (voir Figure 5.4) est un peu différente en ceci que le mouvement est contrôlé par les actions de l'utilisateur. Nous pourrions utiliser la souris pour définir la position de la tourelle, mais le jeu en deviendrait presque trop facile. Il suffirait d'un mouvement de poignet pour passer d'un côté à l'autre de l'écran.

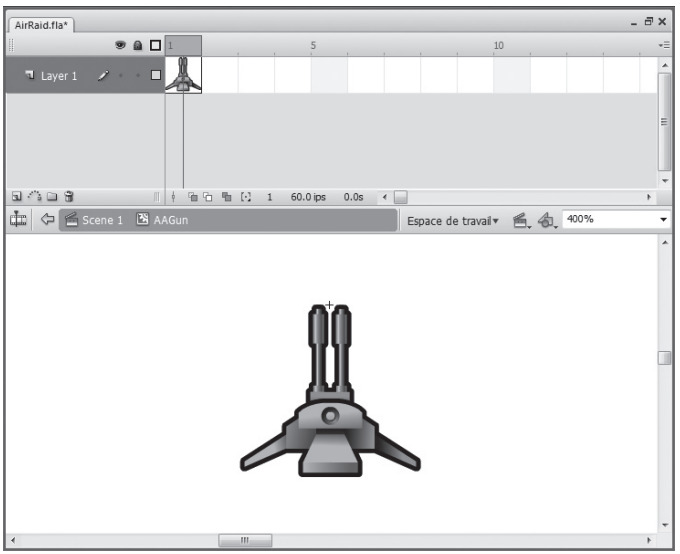


Figure 5.4
La tourelle anti-DCA est positionnée de manière que son point d'alignement se trouve au bout du canon.

Au lieu de cela, nous utiliserons donc les touches fléchées de gauche et de droite pour déplacer le canon. Comme les avions, nous ferons avancer la tourelle à une vitesse déterminée vers la gauche ou la droite en fonction de la touche enfoncée.

Les touches fléchées seront en fait gérées par la classe de l'animation principale et non par la classe `AAGun`. En effet, le clavier envoie par défaut des événements à la scène et non à un clip particulier.

La classe de l'animation principale aura deux variables, `leftArrow` et `rightArrow`, qui seront positionnées à `true` ou à `false`. La classe `AAGun` examine simplement ces variables pour voir dans quelle direction déplacer le canon si celui-ci doit bouger :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class AAGun extends MovieClip {
        static const speed:Number = 150.0;
        private var lastTime:int; // Temps d'animation
```

```
        public function AAGun() {
            // Emplacement initial du canon
            this.x = 275;
            this.y = 340;

            // Mouvement
            addEventListener(Event.ENTER_FRAME,moveGun);
        }
```

Maintenant que l'emplacement du canon a été défini et que l'écouteur a été ajouté, la fonction `moveGun` s'exécute à chaque image afin de gérer les éventuels mouvements :

```
        public function moveGun(event:Event) {
            // Calculer la différence temporelle
            var timePassed:int = getTimer()-lastTime;
            lastTime += timePassed;

            // Position actuelle
            var newx = this.x;

            // Déplacement vers la gauche
            if (MovieClip(parent).leftArrow) {
                newx -= speed*timePassed/1000;
            }

            // Déplacement vers la droite
            if (MovieClip(parent).rightArrow) {
                newx += speed*timePassed/1000;
            }

            // Vérification des limites
            if (newx < 10) newx = 10;
            if (newx > 540) newx = 540;

            // Repositionnement
            this.x = newx;
        }
    }
}
```

En plus de déplacer le canon, vous remarquerez sous le commentaire "Vérification des limites" deux lignes qui s'occupent de vérifier le nouvel emplacement du canon afin de s'assurer qu'il ne sort pas sur les côtés.

Il vaut maintenant la peine de voir comment la classe principale gère les appuis sur les touches. Dans la fonction constructeur, deux appels à `addEventListener` configurent ce dispositif :

```
stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

Les deux fonctions qui sont appelées définissent les valeurs booléennes de `leftArrow` et `rightArrow` comme il se doit :

```
// Touche enfoncée
public function keyDownFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = true;
    } else if (event.keyCode == 39) {
        rightArrow = true;
    }
}
```

La valeur de `event.keyCode` est un nombre qui correspond à une touche du clavier. La touche 37 est la touche fléchée de gauche et la touche 39, celle de droite. Les touches 38 et 40 sont les touches fléchées du haut et du bas, que nous utiliserons dans d'autres chapitres.

```
// Touche relâchée
public function keyUpFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = false;
    } else if (event.keyCode == 39) {
        rightArrow = false;
    }
}
```

Le mouvement du canon est donc produit par un effort conjoint de la classe principale et de la classe `AAGun`. La classe principale gère l'entrée clavier et la classe `AAGun` s'occupe du mouvement.

Il reste une dernière partie dans la classe `AAGun` qui correspond à la fonction `deleteGun`. Nous ne l'utiliserons que lorsqu'il sera temps de supprimer le canon de la scène afin de passer à l'image de fin de partie (`gameover`) :

```
// Supprimer de l'écran et supprimer les événements
public function deleteGun() {
    parent.removeChild(this);
    removeEventListener(Event.ENTER_FRAME,moveGun);
}
```

Il est important de toujours penser à utiliser `removeEventListener` pour se débarrasser des événements d'image et de minuteur. Sans cela, ces événements continueront à se produire même après que vous aurez pensé avoir supprimé l'objet parent.

Les balles tirées en l'air

Les balles sont probablement les plus simples des objets mouvants. Dans ce jeu, le graphisme qui les représente est en fait une grappe de balles (voir Figure 5.5).

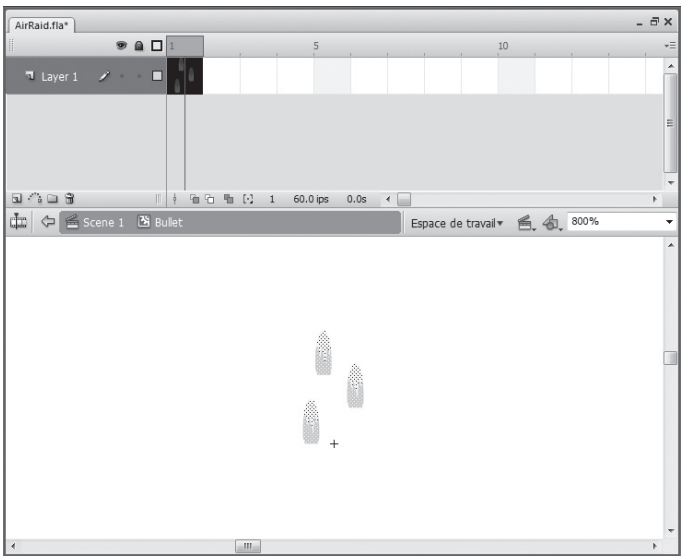


Figure 5.5
Le point d'alignement du groupe de balles se trouve en bas, de sorte qu'en démarrant au niveau du canon elles se trouvent juste au-dessus des canons.

Elles commenceront à l'emplacement du canon et se déplaceront vers le haut jusqu'à atteindre la partie supérieure de l'écran. Nous avons déjà vu dans les classes `Airplane` et `AAGun` tout le code de la classe `Bullet`.

La fonction constructeur accepte une valeur x et une valeur y de départ, ainsi qu'une vitesse. La vitesse sera appliquée verticalement et non horizontalement comme pour les avions :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class Bullet extends MovieClip {
        private var dy:Number; // Vitesse verticale
        private var lastTime:int;

        public function Bullet(x,y:Number, speed: Number) {
            // Définition de la position de départ
            this.x = x;
            this.y = y;
            // Récupérer la vitesse
            dy = speed;
            // Configurer l'animation
            lastTime = getTimer();
            addEventListener(Event.ENTER_FRAME,moveBullet);
        }

        public function moveBullet(event:Event) {
            // Calculer le temps écoulé
            var timePassed:int = getTimer()-lastTime;
            lastTime += timePassed;

            // Déplacer la balle
            this.y += dy*timePassed/1000;

            // La balle a passé le haut de l'écran
            if (this.y < 0) {
                deleteBullet();
            }
        }
```

La fonction `removeBullet`, comme la fonction `removePlane`, se trouvera dans la classe principale. Elle aura à charge de supprimer les balles lorsqu'elles atteignent le haut de l'écran :

```
            // Supprimer la balle de la scène et de la liste des balles
            public function deleteBullet() {
                MovieClip(parent).removeBullet(this);
                parent.removeChild(this);
                removeEventListener(Event.ENTER_FRAME,moveBullet);
            }
        }
    }
```

Pour lancer une balle, le joueur appuie sur la barre d'espace. Nous devons modifier `keyDownFunction` dans la classe `AirRaid` de manière à accepter les espaces et à les passer à une fonction qui gère la création d'une nouvelle balle (`Bullet`) :

```
    // Touche enfoncée
    public function keyDownFunction(event:KeyboardEvent) {
        if (event.keyCode == 37) {
            leftArrow = true;
        } else if (event.keyCode == 39) {
            rightArrow = true;
        } else if (event.keyCode == 32) {
            fireBullet();
        }
    }
```

Le code de touche 32 désigne la barre d'espace. Pour retrouver les correspondances entre les codes et les touches, consultez l'aide de Flash en tapant "Touches du clavier et valeurs de code" dans le champ Rechercher.

La fonction `fireBullet` passe l'emplacement du canon et une vitesse à la nouvelle `Bullet`. Elle ajoute aussi la nouvelle `Bullet` au tableau `bullets` afin de pouvoir en tenir le registre pour la détection de collision ultérieure :

```
    public function fireBullet() {
        var b:Bullet = new Bullet(aagun.x,aagun.y,-300);
        addChild(b);
        bullets.push(b);
    }
```

Maintenant que nous avons des avions, un canon anti-DCA et des objets `Bullet`, il est temps de combiner tous ces éléments avec la classe principale `AirRaid`.

La classe du jeu

La classe `AirRaid` contient toute la logique du jeu. C'est à cet endroit que nous allons créer les objets de jeu initiaux, vérifier les collisions et gérer le score. Une fois que la partie sera lancée, le jeu ressemblera à l'écran présenté à la Figure 5.6.

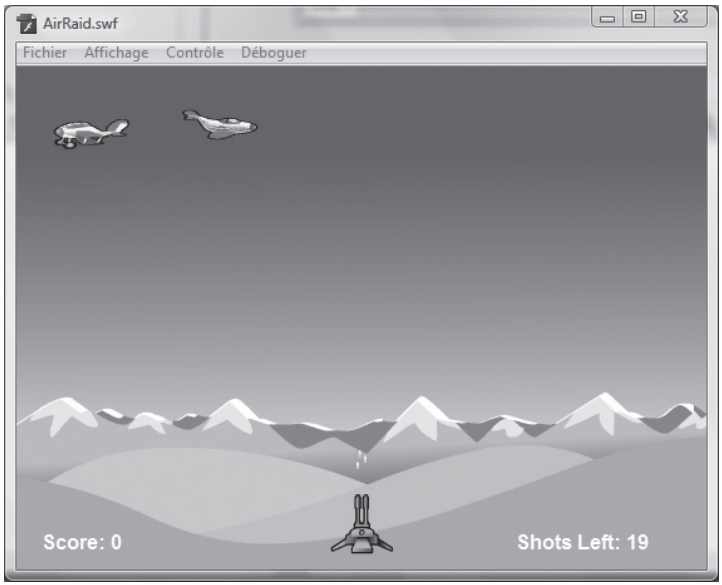


Figure 5.6
Le jeu Air Raid avec un canon anti-DCA, une balle en mouvement et deux avions en vol.

Déclaration de classe

La classe requiert les classes standard que nous avons utilisées jusque-là, et notamment un accès à `getTimer` et aux champs texte :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.Timer;
    import flash.text.TextField;
```

Parmi les variables dont nous avons besoin pour la classe figurent des références au canon et les tableaux qui référencent les avions et les balles que nous avons créés :

```
public class AirRaid extends MovieClip {
    private var aagun:AAGun;
    private var airplanes:Array;
    private var bullets:Array;
```

Les deux variables suivantes sont des valeurs `true` ou `false` qui permettent de surveiller l'usage que fait le joueur des touches fléchées de gauche et de droite. Elles doivent être rendues publiques parce que l'objet `aagun` y accédera afin de déterminer s'il faut se déplacer :

```
    public var leftArrow, rightArrow:Boolean;
```

Vous pouvez inclure plus d'une variable dans une ligne de définition de variable. Cette technique fonctionne bien lorsque vous avez de petits groupes de variables liées et du même type. Les variables `leftArrow` et `rightArrow` sont ici un bon exemple de ce type d'application.

La variable suivante, `nextPlane`, sera un `Timer`. Nous l'utiliserons pour déterminer à quel moment l'avion suivant doit apparaître :

```
    private var nextPlane:Timer;
```

Pour finir, nous avons deux variables de mémorisation du score. La première tient le registre du nombre de coups tirés et la seconde, du nombre d'avions touchés par le joueur :

```
    private var shotsLeft:int;
    private var shotsHit:int;
```

Il n'existe pas de fonction constructeur `AirRaid` pour ce jeu car celui-ci ne commence pas à la première image. Au lieu de cela, nous en aurons une appelée `startAirRaid` qui sera appelée depuis le scénario principal dans l'image `play`.

La fonction commence par définir le nombre de coups restants en le fixant à 20 et par mettre le score à zéro :

```
public function startAirRaid() {
    // Initialisation du score
    shotsLeft = 20;
    shotsHit = 0;
    showGameScore();
```

Ensuite, le canon anti-DCA est créé et ajouté à la scène sous le nom d'occurrence `aagun` :

```
    // Création du canon
    aagun = new AAGun();
    addChild(aagun);
```

Nous devons également créer les tableaux qui contiendront les balles et les avions :

```
        // Création des tableaux d'objets
        airplanes = new Array();
        bullets = new Array();
```

Nous avons besoin de deux écouteurs pour savoir quelle touche fléchée a été enfoncée, l'un pour les événements d'appui sur les touches et l'autre pour les événements de relâchement des touches :

```
        // Écouter le clavier
        stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

Nous avons besoin d'un écouteur événementiel `ENTER_FRAME` pour déclencher la fonction `checkForHits`. Il s'agira de la très importante détection de collision entre les balles et les avions :

```
        // Rechercher les collisions
        addEventListener(Event.ENTER_FRAME,checkForHits);
```

Nous devons maintenant lancer le jeu en plaçant des avions en l'air. C'est le rôle de la fonction `setNextPlane`, que nous examinerons sous peu :

```
        // Faire voler des avions
        setNextPlane();
    }
```

Créer un nouvel avion

Il faudra créer périodiquement de nouveaux avions, à des instants plus ou moins aléatoires. Pour cela, nous utiliserons un `Timer` et déclencherons sous peu un appel à la fonction `newPlane`. La fonction `setNextPlane` créera le `Timer` avec un seul événement et le configurera pour 1 à 2 secondes de délai :

```
    public function setNextPlane() {
        nextPlane = new Timer(1000+Math.random()*1000,1);
        nextPlane.addEventListener(TimerEvent.TIMER_COMPLETE,newPlane);
        nextPlane.start();
    }
```

Lorsque le `Timer` expire, il appelle `newPlane` afin de créer un nouvel avion et de le mettre en piste. Les trois paramètres de l'objet `Airplane` sont calculés aléatoirement avec plusieurs résultats de fonction `Math.random()`. Ensuite, l'avion est créé et ajouté à la scène. Il est également ajouté au tableau `airplanes`.

Pour finir, la fonction `setNextPlane` est appelée de nouveau afin de planifier l'avion suivant :

```
    public function newPlane(event:TimerEvent) {
        // Côté, vitesse et altitude aléatoires
        if (Math.random() > .5) {
            var side:String = "left";
        } else {
```

```
            side = "right";
        }
        var altitude:Number = Math.random()*50+20;
        var speed:Number = Math.random()*150+150;

        // Créer un avion
        var p:Airplane = new Airplane(side,speed,altitude);
        addChild(p);
        airplanes.push(p);

        // Définir le temps pour l'avion suivant
        setNextPlane();
    }
```

Détection de collision

La fonction la plus intéressante de tout le jeu est la fonction `checkForHits`. Elle examine toutes les balles et tous les avions afin de déterminer si l'un d'entre eux entre en collision.

Vous remarquerez que nous parcourons les tableaux en boucle à reculons. Nous procédons ainsi pour ne pas nous emmêler les pinceaux lorsque nous supprimons un élément d'un tableau. Si nous progressions vers l'avant, nous pourrions supprimer l'élément 3 du tableau, ce qui ferait de l'ancien élément 4 le nouvel élément 3. Dès lors, en avançant pour rechercher l'élément 4, nous sauterions un élément.

Nous utiliserons `hitTestObject` pour voir si les rectangles d'encadrement des deux clips se chevauchent. Si c'est le cas, nous réaliserons plusieurs choses. Tout d'abord, nous appellerons `planeHit`, qui entamera la destruction de l'avion. Ensuite, nous supprimerons la balle. Nous augmenterons la somme des avions touchés et réafficherons le code du jeu. Enfin, nous cesserons d'examiner les collisions pour cet avion et passerons à la balle suivante dans la liste :

```
// Vérification des collisions
public function checkForHits(event:Event) {
    for(var bulletNum:int=bullets.length-1;bulletNum>=0;bulletNum--){
        for (var airplaneNum:int=airplanes.length-1;airplaneNum>=0;airplaneNum--) {
            if (bullets[bulletNum].hitTestObject(airplanes[airplaneNum])) {
                airplanes[airplaneNum].planeHit();
                bullets[bulletNum].deleteBullet();
                shotsHit++;
                showGameScore();
```

```
            break;
                    }
                }
            }
            if ((shotsLeft == 0) && (bullets.length == 0)) {
                endGame();
            }
        }
```

À la fin de la fonction, nous vérifierons si la partie est terminée. C'est le cas lorsqu'il ne reste plus de munitions et que la dernière balle a quitté l'écran ou a touché un avion.

Gestion de l'entrée clavier

Les deux fonctions suivantes gèrent les appuis sur les touches. Nous les avons rencontrées toutes les deux auparavant :

```
        // Touche enfoncée
        public function keyDownFunction(event:KeyboardEvent) {
            if (event.keyCode == 37) {
                leftArrow = true;
            } else if (event.keyCode == 39) {
                rightArrow = true;
            } else if (event.keyCode == 32) {
                fireBullet();
            }
        }

        // Touche relâchée
        public function keyUpFunction(event:KeyboardEvent) {
            if (event.keyCode == 37) {
                leftArrow = false;
            } else if (event.keyCode == 39) {
                rightArrow = false;
            }
        }
```

Pour créer une nouvelle balle lorsque le joueur appuie sur la barre d'espace, nous créons simplement l'objet et l'alimentons en lui fournissant l'emplacement du canon et la vitesse de la balle (dans le cas présent, 300 pixels par seconde).

Nous ajoutons la balle au tableau `bullets` et soustrayons une unité à `shotsLeft`. Nous mettrons également à jour le score.

Vous remarquerez qu'avant que quoi que ce soit ne se passe nous vérifions `shotsLeft` afin de nous assurer que le joueur possède encore des munitions à tirer. Cette vérification évite que le joueur ne récupère quelques balles supplémentaires à la fin du jeu :

```
// Nouvelle balle créée
public function fireBullet() {
    if (shotsLeft <= 0) return;
    var b:Bullet = new Bullet(aagun.x,aagun.y,-300);
    addChild(b);
    bullets.push(b);
    shotsLeft--;
    showGameScore();
}
```

Autres fonctions

Nous avons maintenant appelé `showGameScore` plusieurs fois. Cette petite fonction place simplement `shotsHit` et `shotsLeft` dans des champs texte sur la scène. Il s'agit non pas de champs texte que nous avons créés dans le code mais de champs que j'ai placés manuellement dans la scène de l'animation d'exemple. Je ne souhaitais pas encombrer cet exemple avec du code `TextField` et `TextFormat` :

```
public function showGameScore() {
    showScore.text = String("Score: "+shotsHit);
    showShots.text = String("Shots Left: "+shotsLeft);
}
```

Si je n'ai pas créé les champs texte dans le code, j'ai cependant besoin de l'instruction `import flash.text.TextField;` au début de la classe. Cette bibliothèque est nécessaire non seulement pour créer les champs texte, mais également pour y accéder.

Les deux fonctions suivantes suppriment simplement un élément d'un tableau. La boucle `for…in` est utilisée pour parcourir le tableau et la commande `splice`, pour supprimer l'élément une fois qu'il est trouvé. La commande `break` est utilisée pour quitter la boucle une fois la correspondance trouvée.

Nous avons besoin d'une fonction pour supprimer un avion du tableau `airplanes` et d'une autre pour supprimer une balle du tableau `bullets` :

```
// Retirer un avion du tableau
public function removePlane(plane:Airplane) {
    for(var i in airplanes) {
```

```
            if (airplanes[i] == plane) {
                airplanes.splice(i,1);
                break;
            }
        }
    }

    // Retirer une balle du tableau
    public function removeBullet(bullet:Bullet) {
        for(var i in bullets) {
            if (bullets[i] == bullet) {
                bullets.splice(i,1);
                break;
            }
        }
    }
```

Nous pourrions utiliser une unique fonction en lieu et place de `removePlane` et de `removeBullet`. Cette unique fonction se verrait passer à la fois le tableau et l'élément à trouver. En utilisant des fonctions séparées, nous prenons des dispositions en prévision du développement futur du jeu, où la suppression des avions et des balles pourrait avoir d'autres effets. Par exemple, la suppression d'un avion pourrait être le signal pour appeler `setNewPlane` au lieu que cette opération se produise juste après qu'un avion eut été créé.

Nettoyage après la partie

Lorsqu'une partie se termine, il reste des éléments de jeu à l'écran. Nous savons que toutes les balles sont parties parce qu'il s'agit d'une condition qui a dû être remplie pour que la partie se termine. Les avions et le canon sont en revanche toujours là.

Nous n'avons pas stocké tous les objets d'affichage dans un unique tableau comme nous l'avions fait pour le jeu de déduction du précédent chapitre. Au lieu de cela, nous les avons placés dans le tableau `airplanes`, la variable `aagun` et le tableau `bullets`, que nous savons être maintenant vide.

Après avoir supprimé les avions et le canon, nous devons également supprimer les écouteurs clavier et l'écouteur événementiel `checkForHits`. Le `Timer nextPlane` doit également être nettoyé. Ensuite, nous pouvons passer à l'image `gameover` sans qu'aucun élément de jeu ne traîne alentour :

```
    // La partie est terminée, supprimer les clips
    public function endGame() {
        // Supprimer les avions
        for(var i:int=airplanes.length-1;i>=0;i--) {
            airplanes[i].deletePlane();
```

```
        }
        airplanes = null;

        aagun.deleteGun();
        aagun = null;

        stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
        removeEventListener(Event.ENTER_FRAME,checkForHits);

        nextPlane.stop();
        nextPlane = null;

        gotoAndStop("gameover");
    }
```

Après cette fonction, vous aurez besoin de deux accolades supplémentaires afin de refermer la classe et le paquetage.

L'un des intérêts de l'utilisation de champs texte créés manuellement dans le scénario principal au lieu de champs ActionScript tient à ce qu'ils resteront pour l'image `gameover`. Le joueur pourra ainsi voir son score dans la dernière image.

Modifier le jeu

L'animation **AirRaid.fla** contient les mêmes scripts d'image et boutons que l'animation **Deduction.fla** du précédent chapitre. L'image `intro` contient un bouton Start et l'image `gameover`, un bouton Play Again. L'image intermédiaire est appelée `play`.

Dans ce jeu, j'ai également ajouté des instructions à l'image `intro`. La Figure 5.7 présente la première image avec ses instructions, son titre, son bouton Start et les champs texte en bas qui seront utilisés dans le jeu.

Ce jeu peut être amélioré en ajoutant plus d'avions ou en améliorant le graphisme des avions. L'arrière-plan et le canon peuvent aussi être redessinés.

Dans le code, vous pouvez faire varier la fréquence d'apparition des avions et leur vitesse de déplacement. Vous pourriez même accélérer ces mouvements à mesure que la partie progresse.

Vous pouvez également changer le nombre de munitions dont dispose le joueur.

D'autres modifications plus radicales pourraient amener à changer entièrement le thème du jeu. Vous pouvez recréer l'un de ces anciens jeux de sous-marin en transformant les avions en bateaux et le canon en périscope de tir. Dans ce cas-là, je ralentirais personnellement la vitesse des balles et utiliserait des graphismes d'arrière-plan élaborés pour donner plus de profondeur à la scène.

Figure 5.7
L'image intro contient des instructions et un bouton Start.

Casse-brique

Codes sources
http://flashgameu.com
A3GPU05_PaddleBall.zip

Air Raid impliquait un simple mouvement à une dimension pour une variété d'objets et des collisions qui conduisaient à détruire les objets. Ce nouveau jeu de casse-brique (en anglais, *Paddle Ball*) inclut des mouvements diagonaux et des collisions qui provoquent des rebonds.

Le casse-brique a été popularisé par l'ancien jeu vidéo Breakout. Des versions de ce jeu apparaissent souvent sur Internet et sont même devenues des standards sur bien des téléphones portables et des iPod.

La version d'origine de Breakout pour Atari, en 1976, a été développée par Steve Jobs et Steve Wozniak, avant qu'ils ne fondent Apple Computer. La conception compacte de puce de Wozniak pour le jeu n'a pas fonctionné avec le processus de fabrication d'Atari et a donc été abandonnée.
Des versions de Breakout sont apparues en série dans de nombreuses versions du système Mac OS. Aujourd'hui encore, elles sont incluses dans la plupart des iPod.

Dans cette version du jeu de casse-brique, le joueur contrôle en bas de l'écran une raquette qu'il peut déplacer à gauche et à droite à l'aide de la souris. Le principal élément actif du jeu est une balle qui peut rebondir sur les côtés, les murs et en haut mais qui file en bas de l'écran si la raquette ne la renvoie pas.

En haut de l'écran se trouve un certain nombre de briques rectangulaires que le joueur doit éliminer en dirigeant la balle vers elle avec la raquette.

Configurer l'animation

L'animation est organisée comme Air Raid et Deduction. La première image s'appelle `intro` et la troisième, `gameover`. Elles contiennent chacune des boutons pour démarrer un nouveau jeu et des instructions dans la première.

L'image du milieu est intitulée `play`. C'est à cet endroit que se trouve le jeu. Nous avons dessiné ici une bordure, un champ texte au milieu pour les messages tels que "Click to Start" (cliquez pour commencer) et un autre en bas à droite pour indiquer au joueur le nombre de balles restantes. La Figure 5.8 présente ces trois éléments sur fond noir.

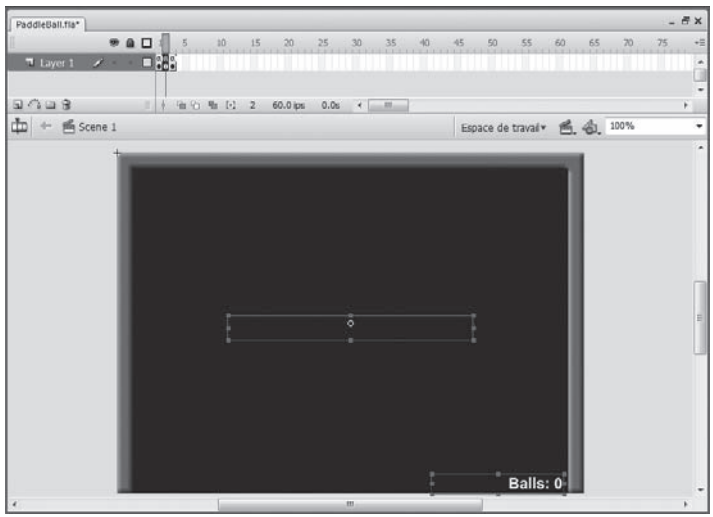


Figure 5.8
Le champ texte du milieu est intitulé `gameMessage` et celui du bas à droite, `ballsLeft`.

La bibliothèque dans ce clip est également un peu plus encombrée que dans les jeux que nous avons examinés jusque-là. La Figure 5.9 présente la bibliothèque avec le nom de classe des clips auxquels nos scripts accéderont.

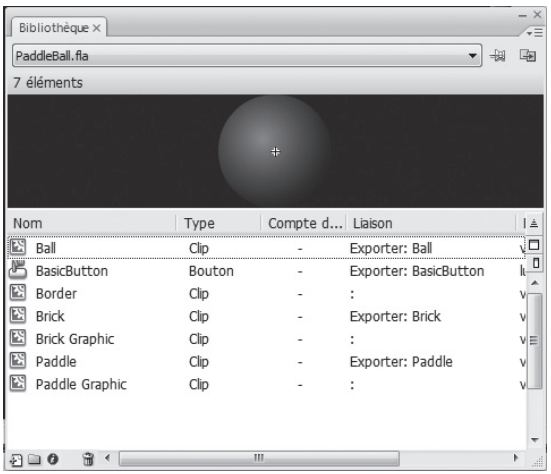
Vous remarquerez qu'il y a un clip `Brick` et un clip `Brick Graphic` ainsi qu'un clip `Paddle` et un clip `Paddle Graphic`. Le second élément de chaque paire est contenu dans le premier. Le clip `Brick Graphic` est ainsi le seul et unique élément dans `Brick`.

L'intérêt de cette approche tient à ce qu'elle nous permet aisément d'appliquer un filtre à ces éléments. Un filtre, comme le filtre Biseau que nous utiliserons ici, peut être appliqué dans l'inspecteur des propriétés du

clip. Comme `Brick` et `Paddle` seront tous deux créés par ActionScript, nous ne pouvons cependant pas leur appliquer ces filtres de cette manière. Voilà pourquoi nous avons un `Brick Graphic` à l'intérieur de `Brick`.

Le clip `Brick Graphic` se verra appliquer le filtre. Ensuite, nous pourrons créer une copie de `Brick` en ActionScript sans le moindre souci.

Figure 5.9
La bibliothèque compte sept éléments, dont la balle, les briques et la raquette.



Nous pourrions appliquer le filtre avec le code ActionScript également, mais il faudrait du code supplémentaire qui ne concerne pas véritablement le jeu. L'autre raison pour laquelle il est préférable de conserver les réglages de filtre en dehors du code ActionScript tient à ce que ces éléments peuvent être laissés au soin d'un artiste graphiste qui travaille avec le programmeur. Il est plus simple pour les artistes de créer des graphismes et d'appliquer des filtres que de dépendre du programmeur pour cela.

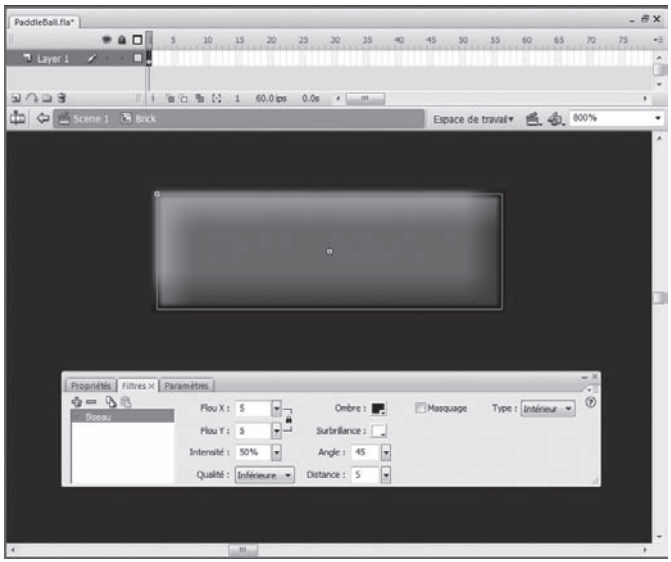
La Figure 5.10 présente le clip `Brick`, avec le clip `Brick Graphic` à l'intérieur. Vous pouvez également voir l'inspecteur des propriétés avec les paramètres de filtre.

Définition de classe

À la différence du jeu Air Raid, celui-ci utilise une unique classe ActionScript pour tout contrôler. Cette classe a besoin de renfort, notamment avec `getTimer`, la classe `Rectangle` et la classe `TextField` :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;
    import flash.geom.Rectangle;
    import flash.text.TextField;
```

Figure 5.10
Un filtre Biseau est utilisé pour transformer des rectangles simples en graphismes plus sophistiqués.



De nombreuses constantes doivent être considérées dans ce jeu. La liste suivante s'explique relativement bien d'elle-même. Elle inclut les positions et dimensions de nombreux éléments dans le jeu, comme la balle, les murs et la raquette :

```
public class PaddleBall extends MovieClip {
    // Constantes environnementales
    private const ballRadius:Number = 9;
    private const wallTop:Number = 18;
    private const wallLeft:Number = 18;
    private const wallRight:Number = 532;
    private const paddleY:Number = 380;
    private const paddleWidth:Number = 90;
    private const ballSpeed:Number = .2;
    private const paddleCurve:Number = .005;
    private const paddleHeight:Number = 18;
```

Les deux seuls objets qui se déplacent dans ce jeu sont la balle et la raquette. En outre, nous avons besoin d'un tableau pour stocker les briques :

```
    // Objets clés
    private var paddle:Paddle;
    private var ball:Ball;

    // Briques
    private var bricks:Array;
```

Nous utiliserons deux variables pour surveiller la vélocité de la balle : `ballDX` et `ballDY`. Nous aurons également besoin d'une variable `lastTime`, comme dans `Air Raid` :

```
// Vélocité de la balle
private var ballDX:Number;
private var ballDY:Number;

// Minuteur de l'animation
private var lastTime:uint;
```

La vélocité est une mesure combinant vitesse et direction. Une variable comme `dx` mesure la vitesse horizontale d'un objet. Mais une combinaison de variables comme `dx` et `dy` mesure à la fois la vitesse et la direction : autrement dit, la vélocité. Sinon un jeu peut enregistrer la vitesse dans une variable (pixels par seconde) et la direction dans une autre (un vecteur ou un angle). Combinées, ces valeurs représentent la vélocité.

Une dernière variable définit le nombre de balles restantes. Il s'agit du premier jeu que nous ayons créé qui donne au joueur un certain nombre de chances, ou de vies, avant que la partie ne soit terminée. Le joueur a trois balles à utiliser lorsqu'il joue. Lorsqu'il manque la troisième, la partie se termine :

```
// Nombre de balles restantes
private var balls:Number;
```

Il n'y aura pas de fonction constructeur pour ce jeu car nous attendrons la deuxième image pour démarrer. Nous laissons donc de côté la fonction `PaddleBall`.

Démarrer le jeu

Lorsque le jeu est démarré, la raquette, les briques et la balle sont créées. La création du motif de briques est déléguée à une autre fonction. Nous l'examinerons ensuite.

Le nombre de balles est fixé à trois et le message de jeu initial est placé dans le champ texte. `lastTime` est également positionné à zéro. Nous ne procédions pas ainsi auparavant, puisque nous lui attribuions la valeur de `getTimer`. Je m'expliquerai sur ce point lorsque nous en viendrons aux fonctions d'animation qui utilisent `lastTime`.

Deux écouteurs sont définis. Le premier appelle simplement la fonction `moveObjects` à chaque image. Le second est un écouteur événementiel placé dans la scène pour capturer les clics de souris. Nous allons demander au joueur de cliquer pour démarrer (message "Click to Start").

Nous devons donc capturer ce clic et l'utiliser en exécutant `newBall` :

```
public function startPaddleBall() {

    // Créer la raquette
    paddle = new Paddle();
    paddle.y = paddleY;
    addChild(paddle);

    // Créer les briques
    makeBricks();

    balls = 3;
    gameMessage.text = "Click To Start";

    // Configurer l'animation
    lastTime = 0;
    addEventListener(Event.ENTER_FRAME,moveObjects);
    stage.addEventListener(MouseEvent.CLICK,newBall);
}
```

La fonction `makeBricks` crée une grille de briques. Il y aura huit colonnes en largeur et cinq lignes en hauteur. Nous utiliserons une boucle imbriquée avec des variables `x` et `y` pour créer l'ensemble des quarante briques. Les positions de chaque brique correspondent à des espaces de 60 pixels verticalement et 20 horizontalement, avec un décalage de 65 et 50 pixels :

```
// Créer la collection de briques
public function makeBricks() {
    bricks = new Array();

    // Créer une grille de briques, 5 verticalement et 8 horizontalement
    for(var y:uint=0;y<5;y++) {
        for(var x:uint=0;x<8;x++) {
            var newBrick:Brick = new Brick();
            // Les espacer harmonieusement
            newBrick.x = 60*x+65;
            newBrick.y = 20*y+50;
            addChild(newBrick);
            bricks.push(newBrick);
        }
    }
}
```

Lors de la création de fonctions d'organisation de ce type, ne craignez pas d'effectuer quelques expérimentations avec les nombres afin d'obtenir le résultat désiré. Par exemple, j'ai testé différents nombres dans la fonction `makeBricks` *jusqu'à obtenir une disposition des briques qui ait bonne apparence. J'aurais évidemment pu calculer l'espacement et le décalage sur papier auparavant, mais il était plus simple et plus rapide de procéder à quelques essais avisés.*

ActionScript est un excellent environnement pour l'expérimentation et la découverte. Vous n'avez pas besoin de prévoir les moindres détails avant de taper la première ligne de code.

L'un des avantages liés au fait de déléguer la création des briques à sa propre fonction tient à ce que vous pouvez la remplacer par une fonction qui produit différents motifs de briques. Cette fonction pourrait même récupérer dans une base de données un schéma définissant la disposition des différents murs de briques.

Tous les changements que vous apportez au motif sont isolés dans un unique appel à `makeBricks`. Il est donc aussi aisé de travailler avec un second programmeur qui s'occupe de la disposition des briques pendant que vous travaillez sur la logique du jeu lui-même.

La Figure 5.11 présente le jeu au début, avec la balle, la raquette et les briques. Vous pouvez également voir les murs, qui ont un simple rôle esthétique. La balle rebondira sur les murs invisibles que nous avons créés en définissant `wallLeft`, `wallRight` et `wallTop`.

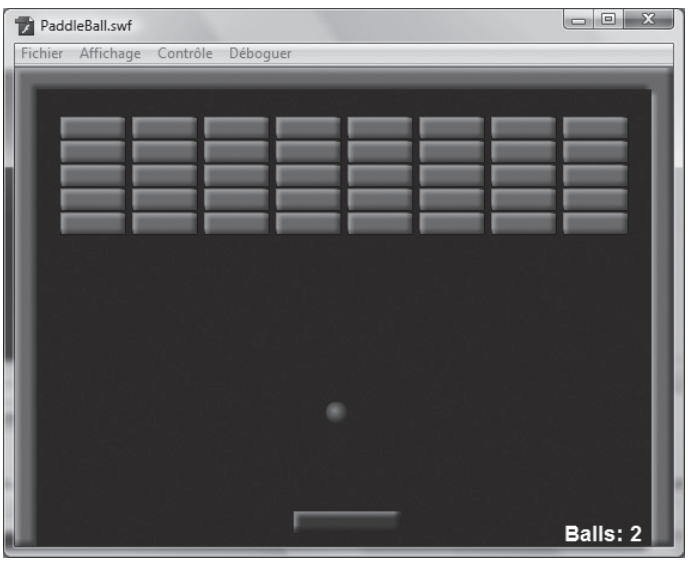


Figure 5.11
Tous les éléments de jeu apparaissent au tout début du jeu.

Lancer une nouvelle balle

Lorsque le jeu commence, il n'y a pas de balle. Au lieu de cela, le message "Click to Start" apparaît et le joueur doit cliquer sur la scène. Cette action appelle `newBall`, qui crée un nouvel objet `Ball` et définit sa position et ses propriétés associées.

Pour commencer, `newBall` s'assure que `ball` vaut `null`. Cette précaution empêche l'utilisateur de cliquer sur l'écran pendant que la balle est déjà en jeu.

Ensuite, le champ texte `gameMessage` est effacé :

```
public function newBall(event:Event) {
    // Ne pas exécuter ce code s'il y a déjà une balle
    if (ball != null) return;

    gameMessage.text = "";
```

Une nouvelle balle est créée au centre exact de l'écran, en utilisant le point à mi-chemin entre `wallLeft` et `wallRight` et entre `wallTop` et la position verticale de la raquette :

```
    // Créer la balle, la centrer
    ball = new Ball();
    ball.x = (wallRight-wallLeft)/2+wallLeft;
    ball.y = 200;//(paddleY-wallTop)/2+wallTop;
    addChild(ball);
```

La vélocité de la balle est définie à la verticale exacte, à la vitesse définie par la constante `ballSpeed` :

```
    // vélocité de la balle
    ballDX = 0;
    ballDY = ballSpeed;
```

La variable `balls` est réduite d'une unité et le champ texte en bas à droite est modifié afin d'afficher le nombre de balles restantes. `lastTime` est en outre repositionné à zéro afin que le chronométreur de l'animation recommence à zéro :

```
    // Utiliser une balle
    balls--;
    ballsLeft.text = "Balls: "+balls;

    // Réinitialiser l'animation
    lastTime = 0;
}
```

La fonction `newBall` sera utilisée au début du jeu et également pour lancer une nouvelle balle en cours de partie.

Animation du jeu et détection des collisions

Jusque-là, le code du jeu a été plutôt simple. Les choses se compliquent cependant lorsque l'on considère les objets mouvants. La balle doit détecter les collisions avec la raquette et avec les briques. Elle doit ensuite réagir à ces collisions de manière appropriée.

L'écouteur événementiel pour `ENTER_FRAME` appelle `moveObjects` à chaque image. Cette fonction délègue ensuite le travail à deux autres fonctions, `movePaddle` et `moveBall` :

```
public function moveObjects(event:Event) {
    movePaddle();
    moveBall();
}
```

Mouvement de la raquette

La fonction `movePaddle` est très simple. Elle positionne simplement la propriété `x` de la raquette en lui attribuant l'emplacement de `mouseX`. Elle utilise cependant aussi `Math.min` et `Math.max` pour restreindre l'emplacement aux côtés gauche et droit de la scène.

Les propriétés `mouseX` et `mouseY` retournent l'emplacement du curseur par rapport à l'objet d'affichage courant. Dans le cas présent, il s'agirait de la classe principale, qui équivaut à la scène. Si nous examinions `mouseX` et `mouseY` depuis l'intérieur d'une classe de clip, nous serions contraints d'ajuster les résultats ou d'examiner plutôt `stage.mouseX` et `stage.mouseY`.

```
public function movePaddle() {
    // Faire correspondre la valeur horizontale avec la souris
    var newX:Number = Math.min(wallRight-paddleWidth/2,
        Math.max(wallLeft+paddleWidth/2,
            mouseX));
    paddle.x = newX;
}
```

Mouvement de la balle

C'est `moveBall`, la fonction qui déplace la balle, qui obtient la part du lion en matière de code. Le mouvement de base est comparable à celui des objets mouvants du jeu Air Raid. En revanche, les collisions sont bien plus complexes.

La fonction commence par une vérification de `ball` afin de s'assurer qu'elle ne vaut pas `null`. Si c'est le cas, nous sommes entre deux balles et le reste peut être ignoré :

```
public function moveBall() {
    // Ne pas exécuter ce code si entre deux balles
    if (ball == null) return;
```

Rappelez-vous que nous avons initialisé la variable `lastTime` à zéro au lieu de lui attribuer la valeur de `getTimer`. Nous avons procédé ainsi afin que le temps requis pour créer les objets du jeu et dessiner l'écran la première fois ne soit pas utilisé pour déterminer la quantité de la première étape d'animation. Par exemple, s'il faut 500 millisecondes pour que le jeu commence, `getTimer()` moins `lastTime` vaudra 500 millisecondes ou plus. La balle ferait ainsi un sacré bond avant que le joueur n'ait même la chance de pouvoir réagir.

L'une des raisons pour lesquelles ce jeu prend une demi-seconde environ pour commencer tient à son usage des filtres Biseau. Cette opération ralentit le début du jeu pour générer les modifications de la représentation visuelle des briques et de la raquette. Elle ne ralentit en revanche pas le jeu ensuite.

En faisant commencer `lastTime` à zéro, nous pouvons le reconnaître ici dans la fonction d'animation et obtenir une nouvelle valeur pour `lastTime`. Cela signifie que, la première fois dans la fonction, `timePassed` vaudra en toute vraisemblance zéro. Ce n'est pas un problème et nous pouvons en revanche ainsi nous assurer que le minuteur de l'animation ne tourne pas tant que nous n'en sommes pas au point d'appeler `moveBall` la première fois :

```
// Calcul du nouvel emplacement pour la balle
if (lastTime == 0) lastTime = getTimer();
var timePassed:int = getTimer()-lastTime;
lastTime += timePassed;
var newBallX = ball.x + ballDX*timePassed;
var newBallY = ball.y + ballDY*timePassed;
```

Détection des collisions

Pour démarrer notre détection des collisions, nous récupérerons le rectangle de la balle. En fait, nous allons obtenir deux versions différentes du rectangle : le rectangle actuel de la balle, appelé `oldBallRect`, et le rectangle de la balle si elle devait terminer son mouvement sans contrainte, appelé `newBallRect`.

L'objet `Rectangle` est un parfait exemple d'objet capable d'apporter des compléments d'information à partir des données dont vous l'alimentez. Vous lui fournissez une position `x` et `y` et une largeur et une hauteur, mais vous pouvez lui demander des informations interprétées comme les bords supérieur, inférieur, gauche et droit du rectangle. Vous pouvez obtenir des objets `Point` pour les coins (par exemple, `bottomRight` pour le coin inférieur droit). Nous utiliserons les côtés supérieur, inférieur, gauche et droit du rectangle dans nos calculs.

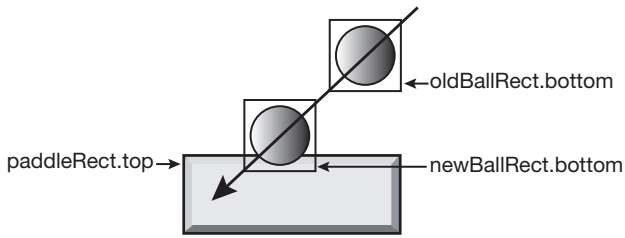
Notre manière de calculer `oldBallRect` et `newBallRect` consiste à utiliser les positions `x` et `y`, plus ou moins le rayon de la balle (`ballRadius`). Par exemple, `ball.x-ballRadius` nous donne la position `x` et `ballRadius*2` nous donne la largeur.

Nous calculons le rectangle de la raquette (`paddleRect`) de la même manière :

```
var oldBallRect = new Rectangle(ball.x-ballRadius,
    ball.y-ballRadius, ballRadius*2, ballRadius*2);
var newBallRect = new Rectangle(newBallX-ballRadius,
    newBallY-ballRadius, ballRadius*2, ballRadius*2);
var paddleRect = new Rectangle(paddle.x-paddleWidth/2,
    paddle.y-paddleHeight/2, paddleWidth, paddleHeight);
```

Maintenant que nous avons ces trois rectangles à disposition, nous pouvons les utiliser pour voir si la balle a touché la raquette. Cela se produit lorsque le bas de la balle franchit le haut de la raquette. Il est cependant plus dur qu'il n'y paraît de déterminer ce moment. Nous ne souhaitons pas simplement savoir si le bas de la balle a passé le bas de la raquette. Nous souhaitons savoir que cela vient tout juste de se passer, exactement à cette étape de l'animation. La véritable question à poser est donc la suivante : le bas de la balle a-t-il franchi le haut de la raquette et le bas de la balle se trouvait-il précédemment au-dessus du haut de la raquette ? Si ces deux conditions sont réunies, la balle vient juste de franchir la raquette. La Figure 5.12 illustre ce problème de manière graphique.

Figure 5.12
Le diagramme présente la balle à l'image précédente et la balle telle qu'elle se positionnerait si son mouvement n'était pas freiné. La balle a franchi la raquette et doit être déviée.



Un autre test doit être réalisé. Nous devons savoir si l'emplacement horizontal de la balle correspond à celui de la raquette. Ainsi, si le côté droit de la balle est supérieur au côté gauche de la raquette et si par ailleurs le côté gauche de la balle est inférieur au côté droit de la raquette, nous avons effectivement une collision.

En cas de collision, la balle doit être déviée vers le haut. Il suffit pour cela d'inverser la direction de `ballDY`. En outre, il convient de définir un nouvel emplacement pour la balle. Celle-ci ne peut tout de même pas rester à l'intérieur de la raquette telle qu'elle s'y trouve maintenant (voir Figure 5.12).

La distance au-delà du haut de la raquette est donc calculée et la balle est déviée vers le haut de deux fois cette distance (voir Figure 5.13).

```
// Collision avec la raquette
if (newBallRect.bottom >= paddleRect.top) {
    if (oldBallRect.bottom < paddleRect.top) {
        if (newBallRect.right > paddleRect.left) {
            if (newBallRect.left < paddleRect.right) {
```

```
                // Rebond
                newBallY -= 2*(newBallRect.bottom - paddleRect.top);
                ballDY *= -1;
                // Calcul du nouvel angle
                ballDX = (newBallX-paddle.x)*paddleCurve;
            }
        }
```

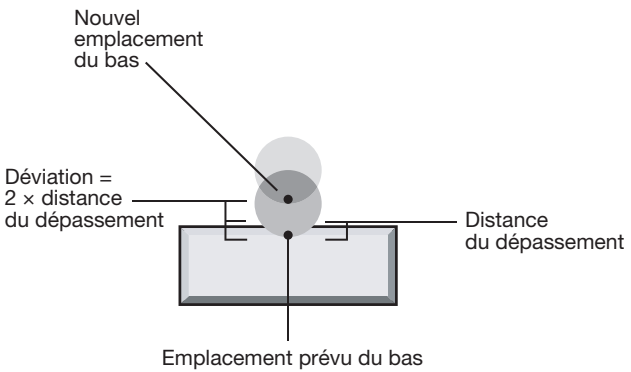


Figure 5.13
La balle avance légèrement dans la raquette et se trouve donc placée à la même distance à l'écart de la raquette.

Si la vitesse verticale de la balle est simplement inversée, la vitesse horizontale `ballDX` se voit pour sa part attribuer une toute nouvelle valeur. Cette valeur est déterminée par la distance à partir du centre de la raquette. Elle est multipliée par une constante `paddleCurve`.

L'idée est ici que le joueur doit diriger la balle ; s'il se contente de toucher la balle avec le plat de la raquette, celle-ci ne prendra pas d'autre angle que celui avec lequel elle a commencé. Le jeu serait impossible à jouer.

L'effet de ce système est que la balle rebondit vers le haut et le bas au centre de la raquette et part avec un angle de plus en plus prononcé à mesure que l'on s'approche de ses extrémités.

Si la balle a passé la raquette, il n'y a pas de retour possible. Nous ne supprimons cependant pas tout de suite la balle. Nous l'autorisons plutôt à continuer jusqu'à ce qu'elle atteigne le bas de l'écran, après quoi elle est finalement supprimée.

L'un des moyens courants de représenter ce principe consiste à faire apparaître une légère courbe sur la partie supérieure de la raquette. Cet effet suggère au joueur l'effet produit lorsque la balle frappe la raquette. Comme ce comportement a été adopté dans tous les jeux de casse-brique depuis, la plupart des joueurs le présupposent quoi qu'il en soit.

S'il s'agit de la dernière balle, la partie est terminée et la fonction `endGame` est appelée. Sinon le champ `gameMessage` est rempli avec le texte `"Click For Next Ball"`. Comme la variable `ball` est positionnée à `null`, la fonction `moveBall` n'agit plus sur rien et la fonction `newBall` accepte le clic suivant comme un déclencheur pour créer une nouvelle balle. Nous pouvons et devons quitter cette fonction maintenant avec une commande `return`. Inutile de vérifier les collisions avec les murs ou les briques si la balle a disparu :

```
        } else if (newBallRect.top > 400) {
            removeChild(ball);
            ball = null;
            if (balls > 0) {
                gameMessage.text = "Click For Next Ball";
            } else {
                endGame();
            }
            return;
        }
    }
```

Ensuite, nous testons les collisions avec les trois murs. Ces vérifications sont plus simples parce que la balle ne doit jamais dépasser l'un d'entre eux. Dans chaque cas, la vitesse verticale ou horizontale est inversée et l'emplacement de la balle est altéré de la même manière que pour les collisions avec la raquette, de sorte que la balle ne se trouve jamais "dans" les murs :

```
    // Collision avec le mur supérieur
    if (newBallRect.top < wallTop) {
        newBallY += 2*(wallTop - newBallRect.top);
        ballDY *= -1;
    }

    // Collisions avec le mur gauche
    if (newBallRect.left < wallLeft) {
        newBallX += 2*(wallLeft - newBallRect.left);
        ballDX *= -1;
    }

    // Collisions avec le mur droit
    if (newBallRect.right > wallRight) {
        newBallX += 2*(wallRight - newBallRect.right);
        ballDX *= -1;
    }
```

Pour calculer la collision avec les briques, nous devons parcourir en boucle toutes les briques et les vérifier une à une. Pour chacune d'entre elles, nous créons un `brickRect` afin de pouvoir accéder aux bords supérieur, inférieur, gauche et droit de la brique aussi facilement qu'avec la balle.

Normalement, lorsque vous souhaitez parcourir en boucle un tableau d'objets afin de vérifier s'il y a une collision, vous devez le faire en sens inverse, afin de pouvoir supprimer des objets dans la liste sans en sauter aucun. Cette fois, nous pouvons avancer normalement car, une fois que la balle entre en collision avec une brique, nous cessons de rechercher des collisions (une seule collision doit en effet se produire à la fois).

Il est aisé de détecter une collision avec une brique, mais déjà plus difficile d'y réagir. Comme nous avons un `Rectangle` pour la balle et un pour la brique, nous pouvons utiliser la fonction `intersects` pour voir si le nouvel emplacement de la balle se trouve à l'intérieur d'une brique.

Si c'est le cas, il faut déterminer quel côté de la brique a été touché. Une série de tests compare les côtés de la balle aux côtés opposés des briques. Lorsqu'un côté chevauché a été trouvé, la balle est déviée dans la direction appropriée et son emplacement est ajusté :

```
// Collision avec les briques
for(var i:int=bricks.length-1;i>=0;i--) {

    // Récupération du rectangle de la brique
    var brickRect:Rectangle = bricks[i].getRect(this);

    // Y a-t-il une collision avec une brique ?
    if (brickRect.intersects(newBallRect)) {

        // La balle frappe le côté gauche ou droit
        if (oldBallRect.right < brickRect.left) {
            newBallX += 2*(brickRect.left - oldBallRect.right);
            ballDX *= -1;
        } else if (oldBallRect.left > brickRect.right) {
            newBallX += 2*(brickRect.right - oldBallRect.left);
            ballDX *= -1;
        }

        // La balle frappe le haut ou le bas
        if (oldBallRect.top > brickRect.bottom) {
            ballDY *= -1;
            newBallY += 2*(brickRect.bottom-newBallRect.top);
        } else if (oldBallRect.bottom < brickRect.top) {
            ballDY *= -1;
            newBallY += 2*(brickRect.top - newBallRect.bottom);
        }
```

Si la balle est entrée en collision avec une brique, cette dernière doit être supprimée. En outre, si le tableau `bricks` est vide, la partie est terminée. Nous souhaitons également utiliser `return` ici parce que, si la partie est terminée, il n'est pas nécessaire de définir l'emplacement de la balle à la fin de cette fonction ou de rechercher d'autres collisions :

```
                // Supprimer la brique
                removeChild(bricks[i]);
                bricks.splice(i,1);
                if (bricks.length < 1) {
                    endGame();
                    return;
                }
            }
        }

        // Définir nouvelle position
        ball.x = newBallX;
        ball.y = newBallY;
    }
```

L'un des aspects importants de ce jeu tient à ce qu'il existe deux modes de jeu véritables : le premier concerne le cas où la balle est en jeu et le deuxième, celui où le joueur doit cliquer sur l'écran pour créer une nouvelle balle. Le code détermine le mode en cours en examinant la valeur de `ball`. Si elle vaut `null`, le jeu se trouve dans le deuxième mode.

Fin de partie

La partie se termine dans deux cas de figure : lorsque le joueur perd la dernière balle ou lorsque la balle atteint la dernière brique.

Comme dans Air Raid, nous utilisons la fonction `endGame` pour nettoyer tous les clips restants. Nous positionnons également les références à ces clips à `null` afin que le lecteur Flash puisse les supprimer de la mémoire.

Il est important de s'assurer que la balle n'est pas déjà partie car elle le sera si la fonction `endGame` est appelée lorsque la dernière balle est perdue.

Nous devons également supprimer les écouteurs, à la fois celui qui appelle `moveObjects` à chaque image et celui qui écoute les clics de souris :

```
    function endGame() {
        // Supprimer la raquette et les briques
        removeChild(paddle);
        for(var i:int=bricks.length-1;i>=0;i--) {
            removeChild(bricks[i]);
```

```
        }
        paddle = null;
        bricks = null;
        // Supprimer la balle
        if (ball != null) {
            removeChild(ball);
            ball = null;
        }

        // Supprimer les écouteurs
        removeEventListener(Event.ENTER_FRAME,moveObjects);
        stage.removeEventListener(MouseEvent.CLICK,newBall);

        gotoAndStop("gameover");
    }
```

À la fin du code, n'oubliez pas les accolades fermantes afin de refermer la classe et le paquetage.

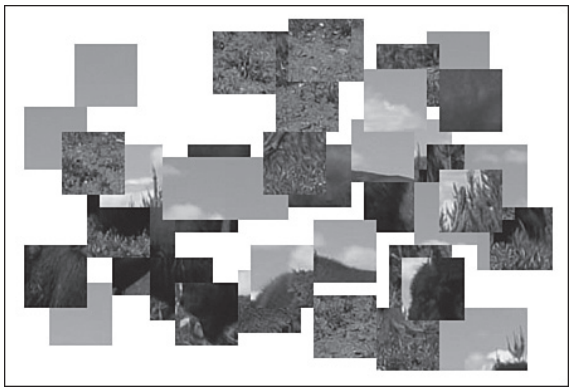
Modifier le jeu

Ce jeu a vraiment besoin de sons. Vous pouvez les ajouter facilement en utilisant les exemples du jeu de Memory du Chapitre 3. Un bon départ pourrait consister à en proposer un pour la frappe sur la raquette et un autre lorsque la balle atteint une brique. Il serait aussi intéressant d'en émettre lorsque la balle touche les murs ou lorsque la balle est manquée.

Il pourrait également être intéressant d'afficher des briques de couleurs différentes, par exemple en utilisant plusieurs images dans le clip `Brick Graphic` puis en accédant chaque fois à l'image appropriée. Chaque ligne pourrait se voir ainsi attribuer sa propre couleur.

Le calcul du score est intéressant, mais il ne témoigne pas clairement d'une réelle progression dans le jeu. Il fonctionnerait mieux si vous pouviez créer plusieurs niveaux. Les autres niveaux pourraient être configurés en augmentant progressivement la vitesse de la balle ou en présentant d'autres dispositions pour les briques.

Lorsque le joueur a détruit toutes les briques, il serait intéressant de faire disparaître la balle et d'afficher un message invitant le joueur à cliquer pour accéder au niveau suivant. En cliquant, le joueur ferait ainsi apparaître non seulement une nouvelle balle mais également un nouveau groupe de briques.



6

Puzzles d'images et puzzles coulissants

Au sommaire de ce chapitre :

- Manipuler des images bitmap
- Jeu de puzzle coulissant
- Jeu de puzzle classique

Il existe toute une gamme de jeux qui utilisent des photographies ou des dessins détaillés. Les puzzles sont assez nouveaux dans l'univers des ordinateurs car il a fallu attendre la moitié des années 1990 pour que les ordinateurs grand public soient dotés de capacités graphiques suffisantes pour afficher des images de qualité.

Flash peut importer un certain nombre de formats d'image différents. Il ne se contente pourtant pas de les importer. Vous pouvez en fait accéder aux données bitmap et manipuler les images. On peut ainsi découper des images et les utiliser dans des puzzles.

Flash prend en charge les formats de fichier JPG, GIF et PNG. Le format JPG est idéal pour les photographies parce qu'il compresse bien les données d'image et permet de déterminer la quantité de compression à utiliser au moment de créer le fichier. Le format GIF est un autre format compressé qui convient mieux aux dessins incluant un nombre limité de couleurs. Le format PNG propose une très bonne compression et une excellente qualité à pleine résolution. Tous ces formats peuvent être créés par Fireworks ou Photoshop d'Adobe, qui sont fournis avec Flash dans certains packs d'Adobe.

Examinons rapidement les notions fondamentales liées à l'importation et à la manipulation des images. Ensuite, nous étudierons deux jeux qui utilisent des pièces de jeu provenant d'images importées et découpées en morceaux.

Manipuler des images bitmap

Codes sources

http://flashgameu.com

A3GPU06_Bitmap.zip

Avant de pouvoir s'amuser avec une image bitmap, il faut commencer par l'importer. Vous pouvez également utiliser une image bitmap dans la bibliothèque en lui attribuant simplement un nom de classe et en y accédant de cette manière. L'importation d'une image bitmap externe est cependant bien plus souple pour presque tous les usages.

Charger une image bitmap

L'objet `Loader` est une version spéciale du `Sprite` qui récupère ses données à partir d'une source externe. Vous aurez besoin de l'apparier avec un `URLRequest`, qui gère l'accès aux fichiers réseau.

Voici une classe très simple qui charge un unique fichier JPG et le positionne à l'écran. Après avoir créé un nouveau `Loader` et un nouveau `URLRequest`, nous les apparions avec la commande `load`.

Le processus complet ne prend que trois lignes de code. Ensuite, nous utilisons `addChild` pour ajouter l'objet `Loader` à la scène, exactement comme pour un objet d'affichage normal tel qu'un `Sprite` :

```
package {
    import flash.display.*;
    import flash.net.URLRequest;
    public class BitmapExample extends MovieClip {

        public function BitmapExample() {
            var loader:Loader = new Loader();
            var request:URLRequest = new URLRequest("myimage.jpg");
            loader.load(request);
            addChild(loader);
        }
    }
}
```

La Figure 6.1 présente une image bitmap chargée de cette manière. Elle est positionnée dans le coin supérieur gauche de l'écran. Comme l'objet `Loader` agit à la manière d'un objet d'affichage normal, nous pouvons également définir sa position `x` et `y` de manière à la centrer à l'écran ou la situer où nous le voulons.

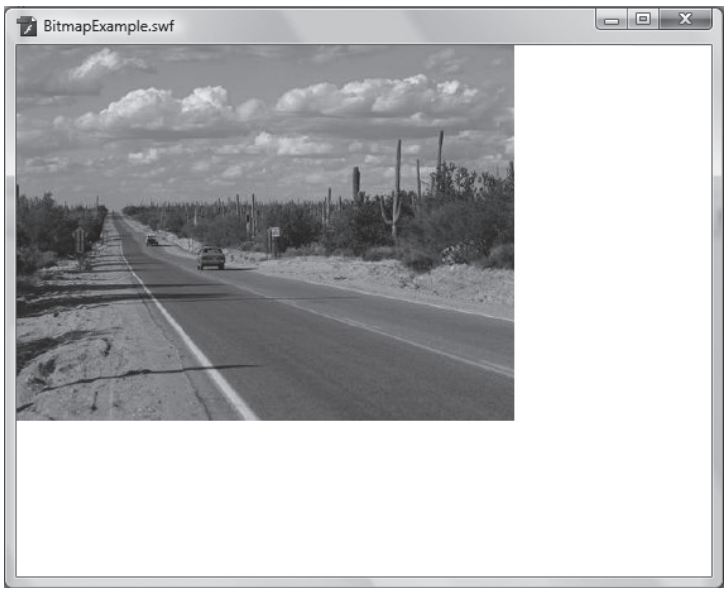


Figure 6.1
Cette image bitmap a été chargée à partir d'un fichier externe mais se comporte maintenant comme un objet d'affichage normal.

Si `URLRequest` est au départ destiné au travail sur les serveurs Web, il fonctionnera tout aussi bien sur votre disque dur pour vos tests.

Ce mécanisme peut être utile pour charger des images d'introduction ou d'instruction dans un jeu. Par exemple, le logo de votre entreprise peut apparaître dans l'écran d'introduction en utilisant ce simple jeu de commandes. Au lieu d'incorporer votre logo dans chaque jeu Flash, vous pouvez avec un unique fichier **logo.png** utiliser un `Loader` et un `URLRequest` pour l'importer et l'afficher. Dès lors, vous n'aurez plus qu'à modifier directement votre fichier **logo.png** pour que tous vos jeux utilisent le nouveau logo.

Décomposer une image bitmap en morceaux

Le chargement et l'affichage d'une simple image bitmap sont utiles, mais nous devons encore nous plonger dans les données et extraire les pièces du puzzle de l'image pour créer les jeux du reste de ce chapitre.

La première modification que nous devons apporter à l'exemple simple précédent consiste à reconnaître le moment auquel l'image bitmap a fini de se charger et où nous pouvons commencer à la manipuler. Nous utiliserons pour cela un écouteur `Event.COMPLETE`. Nous l'ajouterons à l'objet `Loader` et pourrons ensuite placer tout notre code de manipulation dans la fonction `loadingDone` qu'il appellera.

En plus d'`Event.COMPLETE`, vous pouvez également obtenir le rapport d'avancement du téléchargement d'une image. Examinez la documentation Flash concernant `URLRequest` afin d'obtenir d'autres exemples de suivi du chargement et découvrir même des moyens de capturer et de gérer les erreurs.

Voici le début de la nouvelle classe. Nous aurons besoin de la classe `flash.geom` par la suite. J'ai également placé le code de chargement dans sa propre fonction appelée `loadBitmap`, afin qu'il soit aisé à reporter dans les autres jeux du chapitre :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.net.URLRequest;
    import flash.geom.*;

    public class BitmapExample extends MovieClip {

        public function BitmapExample() {
            loadBitmap("testimage.jpg");
        }
```

```
// Récupérer l'image à partir d'une source externe
public function loadBitmap(bitmapFile:String) {
    var loader:Loader = new Loader();
    loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadingDone);
    var request:URLRequest = new URLRequest(bitmapFile);
    loader.load(request);
}
```

Lorsque l'image est complètement chargée, nous appelons la fonction `loadingDone`. Celle-ci s'occupe en premier lieu de créer un nouvel objet `Bitmap`. Elle prend pour cela les données de `event.target.loader.content` (autrement dit, la propriété `content` de l'objet `Loader` original) :

```
private function loadingDone(event:Event):void {
    // Récupérer les données chargées
    var image:Bitmap = Bitmap(event.target.loader.content);
```

Nous pouvons obtenir la largeur et la hauteur de l'image bitmap en accédant aux propriétés `width` et `height` de la variable `image` maintenant qu'elle contient le contenu. Notre but est d'obtenir la largeur et la hauteur de chacune des pièces du puzzle. Pour cet exemple, nous aurons six colonnes et quatre lignes. La largeur totale divisée par six nous donne donc la largeur de chaque pièce. La hauteur totale divisée par quatre nous donne la hauteur de chaque pièce :

```
// Calculer la largeur et la hauteur de chaque pièce
var pieceWidth:Number = image.width/6;
var pieceHeight:Number = image.height/4;
```

À présent, nous parcourons en boucle les six colonnes et les quatre lignes afin de créer chaque pièce du puzzle :

```
// Parcours en boucle de toutes les pièces
for(var x:uint=0;x<6;x++) {
    for (var y:uint=0;y<4;y++) {
```

La création des pièces du puzzle s'opère en créant d'abord un nouvel objet `Bitmap`. Nous spécifions la largeur et la hauteur pour en créer une. Ensuite, nous utilisons `copyPixels` pour copier une section de l'image d'origine dans la propriété `bitmapData` de la pièce de puzzle.

La commande `copyPixels` prend trois paramètres de base : l'image à partir de laquelle copier les données, le `Rectangle` spécifiant la partie de l'image à obtenir et le `Point`, qui définit l'endroit où le morceau de l'image ira dans la nouvelle image bitmap :

```
// Créer une image bitmap de nouvelle pièce de puzzle
var newPuzzlePieceBitmap:Bitmap = new Bitmap(new BitmapData(pieceWidth,pieceHeight));
newPuzzlePieceBitmap.bitmapData.copyPixels(image.bitmapData,new Rectangle(x*pieceWidth,
  y*pieceHeight,pieceWidth,pieceHeight),new Point(0,0));
```

Notre but en soi n'est pas d'avoir une image bitmap. Nous souhaitons avoir un `Sprite` à afficher à l'écran. Nous en créons donc un nouveau et y ajoutons l'image bitmap. Ensuite, nous ajoutons le `Sprite` à la scène :

```
                // Créer un nouveau sprite et y ajouter les données bitmap
                var newPuzzlePiece:Sprite = new Sprite();
                newPuzzlePiece.addChild(newPuzzlePieceBitmap);

                // Ajouter à la scène
                addChild(newPuzzlePiece);
```

Pour finir, nous devons définir l'emplacement des sprites. Nous souhaitons les positionner à l'écran en fonction de leur colonne et de leur ligne, en ajoutant 5 pixels pour créer un espacement blanc entre les pièces. Nous décalons également les positions horizontale et verticale de toutes les pièces de 20 pixels. La Figure 6.2 présente les vingt-quatre pièces du puzzle.

```
                // Définition de l'emplacement
                newPuzzlePiece.x = x*(pieceWidth+5)+20;
                newPuzzlePiece.y = y*(pieceHeight+5)+20;
            }
        }
    }
```

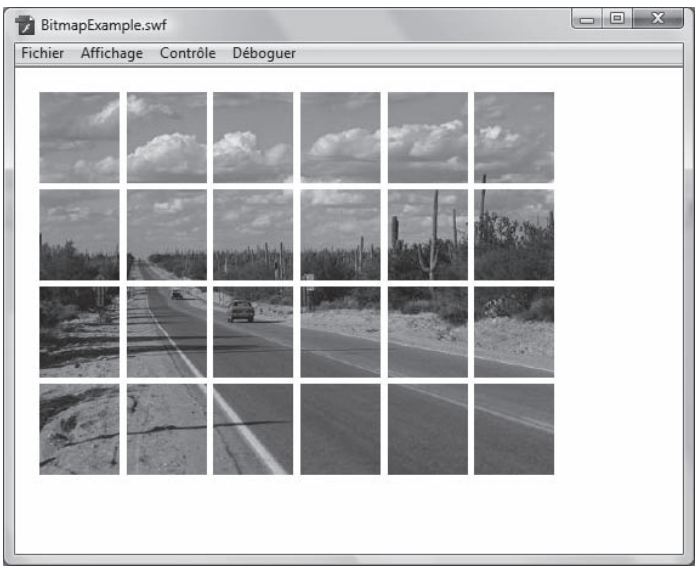


Figure 6.2
L'image importée est décomposée en vingt-quatre pièces qui sont espacées dans l'écran.

Maintenant que nous savons comment créer un ensemble de pièces de puzzle à partir d'une image importée, nous pouvons commencer à créer les jeux qui utilisent ces pièces. Pour commencer, nous allons créer un simple puzzle coulissant. Nous créerons un jeu de puzzle plus complexe par la suite.

Jeu de puzzle coulissant

Codes sources

http://flashgameu.com

A3GPU06_SlidingPuzzle.zip

Il est surprenant qu'un jeu comme le jeu de puzzle coulissant ait même existé avant les ordinateurs. Le jeu physique qui a précédé sa version informatique était composé d'un petit carré en plastique de taille réduite contenant généralement quinze pièces bloquées par de petits rails et des rainures entrecroisés. Il fallait faire glisser l'une des pièces de plastique vers le seizième emplacement inoccupé puis en faire glisser un autre vers le nouvel emplacement vide, et ainsi de suite. Ce processus se poursuivait jusqu'à ce que le puzzle se trouve entièrement reconstitué.

La version physique du jeu utilisait généralement non pas une image mais plutôt des nombres allant de 1 à 15. D'où son nom de baptême : le puzzle-15.

Le problème avec ce jeu en plastique tenait à ce que les carrés coinçaient souvent, ce qui frustrait le joueur, parfois accaparé à débloquer les pièces récalcitrantes. En outre, une fois le puzzle résolu, il fallait déplacer aléatoirement les carrés sans regarder pendant un long moment afin de rétablir une configuration un tant soit peu aléatoire.

La version informatique de ce jeu fonctionne beaucoup mieux. Pour commencer, vous pouvez proposer différents jeux de carrés comme image. Vous pouvez proposer une nouvelle image à chaque fois, en offrant la possibilité aux joueurs de découvrir l'image à mesure que le puzzle se complète. En outre, l'ordinateur peut aléatoirement disposer les pièces du puzzle au début de chaque jeu. Et tout cela sans se pincer les doigts.

Dans notre version du jeu de puzzle coulissant, nous utilisons un nombre variable de pièces découpées dans l'image. En outre, un mélange aléatoire des carrés s'opère au début du jeu. Nous animons également les pièces qui se déplacent d'un endroit à un autre afin qu'elles paraissent glisser. Nous reconnaissons enfin le moment où la solution est trouvée.

Configurer l'animation

Ce jeu utilise la même structure à trois images que nous avons utilisée aux deux derniers chapitres, avec les images `intro`, `play` et `gameover`. Des instructions sont aussi proposées dans la première image.

Le seul graphisme requis est celui de l'image elle-même. Il s'agira d'un fichier d'image JPG externe appelé **slidingimage.jpg**. Nous lui donnerons une taille de 400 pixels par 300.

La taille d'image et la compression doivent être prises en compte lors de la création d'une image pour un jeu de ce type. Les trois images utilisées dans ce chapitre font moins de 34 Ko. Chacune fait 400 × 300 avec une compression JPEG à 80 %. On pourrait aisément produire une image vingt fois plus lourde avec une compression sans perte comme celle des fichiers PNG, mais ce niveau de qualité n'est pas nécessaire et ne ferait que ralentir le temps de téléchargement pour le joueur.

Rappelez-vous que nous allons découper l'image du puzzle puis supprimer la pièce en bas à droite. Le joueur a besoin de cet espace vide pour opérer ses déplacements. Il est donc préférable de choisir une image qui ne contient rien d'important en bas à droite.

Configurer la classe

Le jeu de puzzle coulissant a besoin des classes `URLRequest` et `geom` pour gérer l'image. Nous utiliserons également un objet `Timer` pour l'animation du glissement :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.net.URLRequest;
    import flash.geom.*;
    import flash.utils.Timer;
```

Il y a tout un tas de constantes à définir pour ce jeu, à commencer par l'espacement entre les pièces et le décalage général de toutes les pièces. Nous déciderons également du nombre de pièces à découper dans l'image, qui est ici de 4 × 3 :

```
    public class SlidingPuzzle extends MovieClip {
        // Espacement entre les pièces et décalage
        static const pieceSpace:Number = 2;
        static const horizOffset:Number = 50;
        static const vertOffset:Number = 50;

        // Nombre de pièces
        static const numPiecesHoriz:int = 4;
        static const numPiecesVert:int = 3;
```

Le nombre de colonnes et de lignes dans le puzzle devrait approximativement correspondre aux dimensions de l'image. Dans le cas présent, nous savons qu'il s'agit d'images de 400 pixels sur 300 et nous créons un puzzle de 4 × 3. Les pièces feront donc 100 pixels sur 100. On peut parfaitement créer des pièces rectangulaires, comme un puzzle 4 × 4 avec des pièces de 100 pixels sur 75, mais il semble tout de même préférable de conserver des pièces plutôt carrées.

Pour disposer le puzzle de manière aléatoire au début de la partie, nous allons opérer une série de déplacements aléatoires. Nous reviendrons sur ce point par la suite. En attendant, nous devons stocker le nombre de déplacements aléatoires dans une constante afin de pouvoir le changer facilement :

```
// Étapes du mélange aléatoire
static const numShuffle:int = 200;
```

Les pièces du puzzle glisseront avec fluidité en utilisant un `Timer`. Nous déciderons du nombre d'étapes et de la durée requise pour que le mouvement de glissement s'achève :

```
// Étapes et temps de l'animation
static const slideSteps:int = 10;
static const slideTime:int = 250;
```

La largeur et la hauteur des pièces du puzzle seront calculées en fonction des constantes `numPiecesHoriz` et `numPiecesVert` et de la taille de l'image. Nous récupérerons ces valeurs juste après que l'image aura été chargée :

```
// Taille des pièces
private var pieceWidth:Number;
private var pieceHeight:Number;
```

Il nous faut un tableau pour stocker les pièces du puzzle. Nous y stockerons non pas simplement les références aux nouveaux sprites mais un petit objet contenant l'emplacement de la pièce du puzzle dans le puzzle fini ainsi que la référence au sprite :

```
// Pièces du jeu
private var puzzleObjects:Array;
```

Nous aurons besoin d'un grand nombre de variables pour gérer le jeu et les mouvements. Pour commencer, nous avons `blankPoint`, un objet `Point` indiquant l'emplacement vide dans le puzzle. Lorsque le joueur clique sur une pièce qui lui est adjacente, Flash déplace la pièce pour la loger dans l'espace vide. `slidingPiece` contient une référence à la pièce qui se déplace. `slideDirection` et le `Timer slideAnimation` permettront de paramétrer cette animation :

```
// Suivi des déplacements
private var blankPoint:Point;
private var slidingPiece:Object;
private var slideDirection:Point;
private var slideAnimation:Timer;
```

Lorsque le joueur clique sur le bouton Start, il est conduit à la seconde image, qui appelle `startSlidingPuzzle`. À la différence des fonctions constructeurs d'autres jeux, celle-ci ne fait pas grand-chose. C'est qu'en attendant que l'image soit chargée il n'y pas grand-chose à faire.

La variable `blankPoint` est positionnée au coin inférieur droit en utilisant certaines des constantes. Ensuite, `loadBitmap` est appelée avec le nom du fichier d'image :

```
public function startSlidingPuzzle() {
    // L'espace vide se trouve en bas à droite
    blankPoint = new Point(numPiecesHoriz-1,numPiecesVert-1);

    // Charger l'image bitmap
    loadBitmap("slidingpuzzle.jpg");
}
```

Rappelez-vous que le décompte dans les tableaux et les boucles ActionScript commence à zéro. La pièce en haut à gauche correspond donc à `0,0`. La pièce en bas à droite correspond donc à un de moins que le nombre de pièces en largeur, soit `numPiecesHoriz-1`. Si un puzzle fait quatre pièces en largeur et trois en hauteur, la pièce en bas à droite correspondra donc à `3,2`, soit `numPiecesHoriz-1,numPiecesVert-1`.

Charger l'image

La fonction `loadBitmap` est identique à celle utilisée dans l'exemple précédent de ce chapitre :

```
// Récupérer l'image bitmap à partir d'une source externe
public function loadBitmap(bitmapFile:String) {
    var loader:Loader = new Loader();
    loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadingDone);
    var request:URLRequest = new URLRequest(bitmapFile);
    loader.load(request);
}
```

La fonction `loadingDone` prend dans le cas présent une importance bien plus grande que dans l'exemple antérieur. Maintenant que l'image a été chargée, il est possible d'obtenir la largeur et la hauteur, ce qui nous indique la largeur et la hauteur individuelles de chaque pièce. Nous pouvons définir ces variables puis appeler `makePuzzlePieces` pour opérer le découpage.

Pour finir, `shufflePuzzlePieces` mélange le puzzle et le prépare pour le joueur :

```
// Chargement terminé, procéder au découpage
public function loadingDone(event:Event):void {
    // Créer une nouvelle image pour contenir l'image bitmap chargée
    var image:Bitmap = Bitmap(event.target.loader.content);
    pieceWidth = image.width/numPiecesHoriz;
    pieceHeight = image.height/numPiecesVert;

    // Découpage des pièces du puzzle
    makePuzzlePieces(image.bitmapData);

    // Mélange des pièces
    shufflePuzzlePieces();
}
```

Découper l'image bitmap en morceaux

Si notre précédent exemple découpait l'image en différentes pièces, il n'avait pas à construire tous les objets de données requis pour les rendre utiles dans un jeu. La fonction `makePuzzlePieces` se charge de cette tâche en créant le tableau `puzzleObjects`. Une fois que le sprite de pièce de puzzle est créé et que sa position est définie, la variable temporaire `newPuzzleObject` est créée.

Trois propriétés sont associées à `newPuzzleObject`. La première est `currentLoc`, un objet `Point` qui indique où la pièce de puzzle se trouve actuellement. `0,0` désignera ainsi l'emplacement en haut à gauche et `3,2`, l'emplacement en bas à droite.

De manière similaire, `homeLoc` contient aussi un `Point`. Il s'agit toutefois de l'emplacement original (et final) pour la pièce. Il ne changera pas durant le jeu et fournit un point de référence afin que nous puissions déterminer le moment où chaque pièce est retournée à son emplacement approprié.

L'une des autres approches possibles pourrait être ici de stocker les propriétés `currentLoc` et `homeLoc` des sprites, puis de ne stocker que les sprites dans le tableau. Dans le premier cas, les trois valeurs seraient `puzzleObjects[x].currentLoc`, `puzzleObjects[x].homeLoc` et `puzzleObjects[x].piece`. Dans le dernier, les mêmes données seraient `puzzleObjects[x].currentLoc`, `puzzleObjects[x].homeLoc` et `puzzleObjects[x]` (sans le `.piece` car l'élément dans le tableau est le sprite). Pour ma part, je préfère créer mon propre tableau d'objets afin de m'assurer qu'ActionScript peut rapidement obtenir les informations sans avoir à examiner l'objet `Sprite` tout entier à chaque fois.

Nous avons également une propriété `piece` de `newPuzzleObject`. Cette propriété contient une référence au sprite de la pièce.

Nous stockerons toutes les variables `newPuzzleObject` que nous créons dans le tableau `puzzleObjects` :

```
// Découper l'image bitmap en pièces
public function makePuzzlePieces(bitmapData:BitmapData) {
    puzzleObjects = new Array();
    for(var x:uint=0;x<numPiecesHoriz;x++) {
        for (var y:uint=0;y<numPiecesVert;y++) {
            // Ignorer l'espace vide
            if (blankPoint.equals(new Point(x,y))) continue;

            // Créer l'image bitmap et le sprite de la nouvelle pièce de puzzle
            var newPuzzlePieceBitmap:Bitmap = new Bitmap(new BitmapData(pieceWidth,
              pieceHeight));
            newPuzzlePieceBitmap.bitmapData.copyPixels(bitmapData,
             new Rectangle(x*pieceWidth,y*pieceHeight,
             pieceWidth,pieceHeight),new Point(0,0));
            var newPuzzlePiece:Sprite = new Sprite();
            newPuzzlePiece.addChild(newPuzzlePieceBitmap);
            addChild(newPuzzlePiece);

            // Définir l'emplacement
            newPuzzlePiece.x = x*(pieceWidth+pieceSpace) + horizOffset;
            newPuzzlePiece.y = y*(pieceHeight+pieceSpace) + vertOffset;

            // Créer l'objet à stocker dans le tableau
            var newPuzzleObject:Object = new Object();
            newPuzzleObject.currentLoc = new Point(x,y);
            newPuzzleObject.homeLoc = new Point(x,y);
            newPuzzleObject.piece = newPuzzlePiece;
            newPuzzlePiece.addEventListener(MouseEvent.CLICK,clickPuzzlePiece);
            puzzleObjects.push(newPuzzleObject);
        }
    }
}
```

Chaque pièce de puzzle récupère son propre écouteur événementiel à l'écoute des clics de souris, qui appelle `clickPuzzlePiece`.

À ce stade, toutes les pièces sont en place, mais elles ne sont pas mélangées. Si nous ne les mélangions pas du tout, le jeu commencerait avec le puzzle disposé comme il l'est à la Figure 6.3.

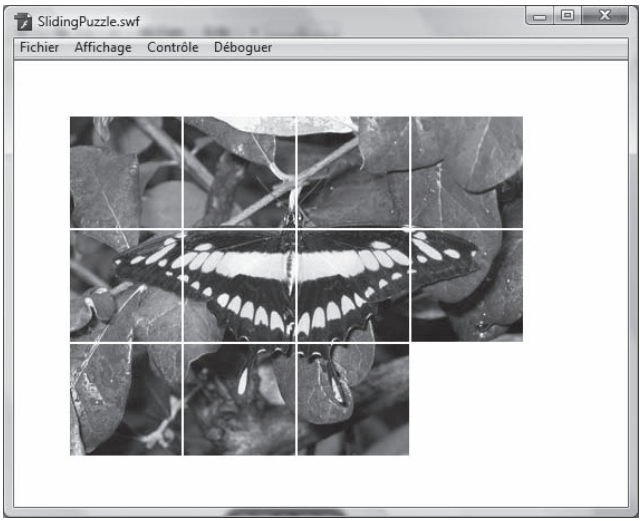


Figure 6.3
Le puzzle coulissant non mélangé. La pièce en bas à droite a été supprimée afin de créer un espace dans lequel les autres pièces peuvent coulisser.

Mélanger les pièces

Une fois que les pièces du puzzle sont en place, il faut les mélanger. L'idée est de créer un "puzzle en désordre" afin que le joueur puisse relever le défi de le redisposer correctement.

L'un des moyens de créer ce désordre consiste à placer toutes les pièces à des emplacements aléatoires. Ce n'est cependant pas ainsi qu'il faut procéder. Si vous placez les pièces aléatoirement, il est assez probable que vous vous retrouviez avec une disposition qui ne permettra jamais de replacer correctement les pièces. La Figure 6.4 présente un tel cas de figure.

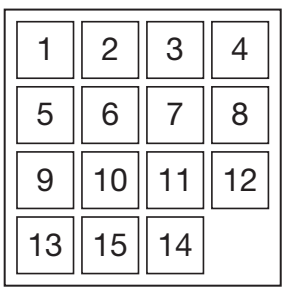


Figure 6.4
En inversant simplement les pièces 15 et 14, nous créons un puzzle impossible à résoudre.

Au lieu de placer aléatoirement chacune des pièces, nous commencerons donc par le puzzle terminé et procéderons à des déplacements aléatoires jusqu'à ce que la grille paraisse complètement aléatoire.

La fonction `shufflePuzzlePieces` opère une boucle et appelle `shuffleRandom` un certain nombre de fois. C'est ici `shuffleRandom` qui se charge du véritable travail :

```
// Créer un certain nombre de déplacements aléatoires
public function shufflePuzzlePieces() {
    for(var i:int=0;i<numShuffle;i++) {
        shuffleRandom();
    }
}
```

Pour créer un déplacement aléatoire, nous allons examiner toutes les pièces du puzzle. Ensuite, nous déterminerons quels déplacements sont possibles et les placerons dans un tableau. Nous choisirons enfin un déplacement au hasard dans ce tableau et procéderons à la modification correspondante.

La clé du mécanisme sera la fonction `validMove`, que nous allons examiner ensuite. La fonction `shuffleRandom`, après avoir sélectionné un déplacement aléatoire, appelle `movePiece`, la même fonction que nous utilisons lorsque le joueur clique pour opérer un déplacement :

```
// Déplacement aléatoire
public function shuffleRandom() {
    // Parcours en boucle pour trouver les déplacements valides
    var validPuzzleObjects:Array = new Array();
    for(var i:uint=0;i<puzzleObjects.length;i++) {
        if (validMove(puzzleObjects[i]) != "none") {
            validPuzzleObjects.push(puzzleObjects[i]);
        }
    }
    // Sélection d'un déplacement aléatoire
    var pick:uint = Math.floor(Math.random()*validPuzzleObjects.length);
    movePiece(validPuzzleObjects[pick],false);
}
```

La fonction `validMove` prend en paramètre une référence à un `puzzleObject`. En utilisant la propriété `currentLoc` de cette pièce de puzzle, elle peut déterminer si la pièce se trouve à côté de l'emplacement vide.

Tout d'abord, elle regarde au-dessus de la pièce de puzzle. Dans ce cas, les valeurs `x` de la pièce et de l'emplacement vide doivent se correspondre. Si c'est effectivement le cas, c'est au tour des positions verticales (`y`) d'être comparées. `blankPoint.y` doit faire un de moins que `currentLoc.y` pour le `puzzleObject`. Si tout cela se confirme, la valeur `"up"` est retournée, qui indique à la fonction appelant `validMove` que cette pièce possède effectivement un déplacement valide : `"up"` (vers le haut).

Vous remarquerez que la déclaration de la fonction `validMove` stipule qu'elle retourne une chaîne. C'est le sens de la portion `": String"` dans la première ligne du code qui suit. Il est toujours judicieux d'indiquer le type de données retourné par vos fonctions. Vous aiderez ainsi le lecteur Flash à optimiser ses performances.

Les déplacements vers le bas, la gauche et la droite sont ensuite examinés. Si aucun de ces déplacements ne ne se révèle valide, la valeur `"none"` est retournée, afin d'indiquer qu'il n'existe pas de déplacement valide pour cette pièce du puzzle :

```
public function validMove(puzzleObject:Object): String {
    // L'emplacement vide est-il au-dessus ?
    if ((puzzleObject.currentLoc.x == blankPoint.x) &&
        (puzzleObject.currentLoc.y == blankPoint.y+1)) {
        return "up";
    }
    // L'emplacement vide est-il en dessous ?
    if ((puzzleObject.currentLoc.x == blankPoint.x) &&
        (puzzleObject.currentLoc.y == blankPoint.y-1)) {
        return "down";
    }
    // L'emplacement vide est-il à gauche ?
    if ((puzzleObject.currentLoc.y == blankPoint.y) &&
        (puzzleObject.currentLoc.x == blankPoint.x+1)) {
        return "left";
    }
    // L'emplacement vide est-il à droite ?
    if ((puzzleObject.currentLoc.y == blankPoint.y) &&
        (puzzleObject.currentLoc.x == blankPoint.x-1)) {
        return "right";
    }
    // Aucun déplacement n'est valide
    return "none";
}
```

Après le mélange, la partie commence avec toutes les pièces en désordre, comme le montre la Figure 6.5.

Figure 6.5
Le jeu peut maintenant commencer avec les pièces mélangées.

La question de savoir combien de fois il faut mélanger les pièces n'a pas de réponse définitive. J'ai personnellement opté pour deux cents déplacements aléatoires, car cela semblait convenir. Si vous choisissez moins, la solution sera plus facile. Si vous choisissez un nombre de déplacements trop élevé, vous constaterez une pause au début du jeu pendant l'opération de mélange.

Réagir aux clics du joueur

Lorsque le joueur clique, la fonction `clickPuzzlePiece` s'exécute. L'événement qui lui est passé possède un `currentTarget` qui correspond à une pièce dans la liste `puzzleObjects`. Une boucle rapide permet de trouver la pièce correspondante et la fonction `movePiece` est appelée :

```
// L'utilisateur a cliqué sur une pièce du puzzle
public function clickPuzzlePiece(event:MouseEvent) {
    // Trouver la pièce cliquée et la déplacer
    for(var i:int=0;i<puzzleObjects.length;i++) {
        if (puzzleObjects[i].piece == event.currentTarget) {
            movePiece(puzzleObjects[i],true);
            break;
        }
    }
}
```

Vous remarquerez que, lorsque la fonction `shuffleRandom` a appelé `movePiece`, elle a utilisé `false` comme second paramètre. Lorsque `clickPuzzlePiece` a appelé `movePiece`, elle a en revanche utilisé `true` comme second paramètre.

Le second paramètre est `slideEffect`, qui possède la valeur `true` ou la valeur `false`. Si la valeur est `true`, un objet `Timer` est créé afin de déplacer progressivement la pièce en un court instant. Si la valeur est `false`, la pièce se déplace immédiatement. Nous souhaitions que les pièces se déplacent immédiatement dans le cas du mélange mais, quand le joueur est en pleine partie, qu'il puisse profiter de cette animation.

La décision n'est en fait pas réalisée à l'intérieur de `movePiece`. `movePiece` se contente en réalité d'appeler `validMove` et de déterminer si le déplacement se fait vers le haut, le bas, la gauche ou la droite. Ensuite, elle appelle `movePieceInDirection` avec les mêmes valeurs `puzzleObject` et `slideEffect`, ainsi que de nouvelles valeurs `dx` et `dy` en fonction de la direction du mouvement.

La fonction `movePiece` utilise la structure `switch` d'ActionScript pour opérer le branchement vers un bloc de code parmi quatre possibles. L'instruction `switch` s'apparente à une série d'instructions `if…then` mais ne requiert la variable testée que dans la ligne `switch`. Chaque branche démarre avec le mot-clé `case` et la valeur correspondant à ce cas. Chaque branche doit se terminer par une commande `break`.

```
// Déplacement d'une pièce dans l'espace vide
public function movePiece(puzzleObject:Object, slideEffect:Boolean) {
    // Déterminer le sens vers l'espace vide
    switch (validMove(puzzleObject)) {
        case "up":
            movePieceInDirection(puzzleObject,0,-1,slideEffect);
            break;
        case "down":
            movePieceInDirection(puzzleObject,0,1,slideEffect);
            break;
        case "left":
            movePieceInDirection(puzzleObject,-1,0,slideEffect);
            break;
        case "right":
            movePieceInDirection(puzzleObject,1,0,slideEffect);
            break;
    }
}
```

La fonction `movePieceInDirection` change instantanément la propriété `currentLoc` de la pièce et la variable `blankPoint`. Ce mécanisme permet de gérer le cas où le joueur réalise un autre mouvement avant que l'animation ne soit terminée. La pièce et le `blankPoint` se trouvent ainsi déjà véritablement au bon emplacement. L'animation n'a dès lors de vocation qu'esthétique :

```
// Déplacer la pièce dans l'espace vide
public function movePieceInDirection(puzzleObject:Object,
dx,dy:int, slideEffect:Boolean) {
    puzzleObject.currentLoc.x += dx;
    puzzleObject.currentLoc.y += dy;
    blankPoint.x -= dx;
    blankPoint.y -= dy;
```

S'il doit y avoir une animation, la fonction `startSlide` est appelée pour la configurer. Sans cela, la pièce du puzzle est déplacée immédiatement vers le nouvel emplacement :

```
    // Animer ou non
    if (slideEffect) {
        // Lancer animation
        startSlide(puzzleObject,dx*(pieceWidth+pieceSpace),
        dy*(pieceHeight+pieceSpace));
    } else {
        // Aucune animation, déplacer seulement
        puzzleObject.piece.x =
puzzleObject.currentLoc.x*(pieceWidth+pieceSpace) + horizOffset;
        puzzleObject.piece.y =
puzzleObject.currentLoc.y*(pieceHeight+pieceSpace) + vertOffset;
    }
}
```

Animer le glissement

Le glissement animé se réalise en lançant un objet `Timer` et en déplaçant la pièce du puzzle par étapes. Selon les constantes au début de la classe, il doit y avoir dix étapes, espacées chacune de 250 millisecondes.

La fonction `startSlide` commence par configurer certaines variables afin de surveiller l'animation. La pièce du puzzle à déplacer est `slidingPiece`. `slideDirection` est un objet `Point` dont `dx` et `dy` représentent le sens du déplacement. Il s'agit de (1,0), (–1,0), (0,1) ou (0,–1) selon le sens dans lequel se déplace la pièce.

Ensuite, le `Timer` est créé et deux écouteurs lui sont associés. L'écouteur `TimerEvent.TIMER` déplace la pièce tandis que l'écouteur `TimerEvent.TIMER_COMPLETE` appelle `slideDone` pour réaliser l'animation :

```
// Configuration d'un glissement
public function startSlide(puzzleObject:Object, dx, dy:Number) {
    if (slideAnimation != null) slideDone(null);
    slidingPiece = puzzleObject;
    slideDirection = new Point(dx,dy);
    slideAnimation = new Timer(slideTime/slideSteps,slideSteps);
    slideAnimation.addEventListener(TimerEvent.TIMER,slidePiece);
    slideAnimation.addEventListener(TimerEvent.TIMER_COMPLETE,slideDone);
    slideAnimation.start();
}
```

Toutes les 250 millisecondes, la pièce de puzzle se déplace d'une fraction de la distance afin de se rapprocher de sa destination finale.

Vous remarquerez également que la première ligne de `startSlide` correspond à un appel potentiel à `slideDone`. Cet appel s'effectue si une nouvelle animation de glissement est sur le point de commencer alors que la première n'a pas fini de se réaliser. Dans ce cas, la précédente est rapidement terminée par un unique appel à `slideDone` qui place la pièce mouvante à son emplacement final et laisse donc la place pour l'animation du nouveau glissement.

Il s'agit cette fois d'un type d'animation différent de l'animation temporelle utilisée au Chapitre 5. Le mouvement de glissement animé n'est pas ici un élément critique du jeu. Il n'a de vertu qu'esthétique. Il est donc judicieux de le placer en dehors du reste de la logique du jeu, avec son propre minuteur temporaire. Nous n'avons pas à nous soucier de l'homogénéité des performances, car elles n'affectent pas le jeu.

```
// Avancer d'une étape dans le glissement
public function slidePiece(event:Event) {
    slidingPiece.piece.x += slideDirection.x/slideSteps;
    slidingPiece.piece.y += slideDirection.y/slideSteps;
}
```

Lorsque le `Timer` a terminé, l'emplacement de la pièce du puzzle est défini de manière à s'assurer que la pièce se trouve exactement où elle doit être. Le minuteur `slideAnimation` est ensuite supprimé.

C'est également le moment d'appeler `puzzleComplete` afin de voir si chaque pièce se trouve au bon endroit. Si c'est le cas, nous appelons `clearPuzzle` et le scénario principal passe à l'image `gameover` :

```
// Fin du glissement
public function slideDone(event:Event) {
    slidingPiece.piece.x =
slidingPiece.currentLoc.x*(pieceWidth+pieceSpace) + horizOffset;
    slidingPiece.piece.y =
slidingPiece.currentLoc.y*(pieceHeight+pieceSpace) + vertOffset;
    slideAnimation.stop();
        slideAnimation = null;

    // Vérifier si le puzzle est terminé
    if (puzzleComplete()) {
        clearPuzzle();
        gotoAndStop("gameover");
    }
}
```

Fin de partie et nettoyage

Il est très facile de déterminer si la partie est terminée. Il suffit pour cela d'examiner chaque pièce et de comparer `currentLoc` avec `homeLoc`. Grâce à la fonction `equals` de la classe `Point`, cela peut se faire en une étape.

Si toutes les pièces se trouvent dans leur emplacement d'origine, la fonction renvoie la valeur `true` :

```
// Vérifier si toutes les pièces sont en place
public function puzzleComplete():Boolean {
    for(var i:int=0;i<puzzleObjects.length;i++) {
        if (!puzzleObjects[i].currentLoc.equals(puzzleObjects[i].homeLoc)) {
            return false;
        }
    }
    return true;
}
```

Vient ensuite la fonction de nettoyage du jeu. Tous les sprites de pièce de puzzle se débarrassent de leurs événements `MouseEvent.CLICK` et sont supprimés, après quoi le tableau `puzzleObjects` est

réinitialisé à `null`. Comme les pièces du puzzle sont les seuls objets du jeu qui ont été créés, il n'y a rien de plus à faire :

```
// Supprimer toutes les pièces du puzzle
public function clearPuzzle() {
    for (var i in puzzleObjects) {
puzzleObjects[i].piece.addEventListener(MouseEvent.CLICK,
clickPuzzlePiece);
        removeChild(puzzleObjects[i].piece);
    }
    puzzleObjects = null;
}
```

Modifier le jeu

Le jeu est plutôt simple à jouer et peu de variantes sont susceptibles de l'améliorer. Vous pouvez cependant améliorer le programme lui-même en offrant la possibilité de sélectionner l'image de manière dynamique. La page Web pourrait ainsi passer elle-même le nom de l'image. Et vous pourriez proposer un jeu qui utilise différentes images selon la page Web ou selon le jour.

Ce jeu peut également être rendu progressif. Après que chaque puzzle est terminé, un nouveau niveau peut être proposé. Le nom de fichier de l'image et les variables `numPiecesHoriz` et `numPiecesVert` peuvent être passés en paramètres à `startSlidingPuzzle`. Au lieu d'accéder à l'image `gameover` une fois le puzzle terminé, vous pourriez ainsi passer au niveau suivant avec un puzzle plus grand et plus difficile à recomposer.

Puzzle classique

Codes sources
http://flashgameu.com
A3GPU06_JigsawPuzzle.zip

Les puzzles classiques se sont popularisés au XVIIIe siècle. À cette époque, on les construisait en bois et à l'aide d'une véritable scie (d'où leur nom anglais, *jigsaw puzzle* ou puzzle-scie). Aujourd'hui, ces puzzles sont faits en carton avec une presse à découper. Les puzzles modernes peuvent être très sophistiqués. Certains atteignent jusqu'à 24 000 pièces.

Les puzzles pour ordinateur se sont popularisés sur le Web et dans les jeux de CD dès la fin des années 1990. Dans ce chapitre, nous allons créer un jeu de puzzle simple en utilisant des pièces rectangulaires découpées dans une image importée.

Dans notre jeu de puzzle, nous allons découper les pièces comme pour le jeu de puzzle coulissant. Au lieu de les placer dans des endroits particuliers à l'écran, nous les disposerons cependant cette fois de manière complètement aléatoire. Le joueur pourra alors faire librement glisser les pièces à l'écran.

La difficulté principale pour la programmation de ce jeu est d'amener les pièces à se "coller" lorsque le joueur les positionne côte à côte.

La plupart des jeux de puzzle prennent la peine de créer des pièces de puzzle qui possèdent l'apparence des pièces traditionnelles du jeu en carton, avec des embouts entrecroisés. Cette fonctionnalité esthétique peut être réalisée avec Flash mais uniquement en utilisant un certain nombre d'outils de dessin vectoriel et de manipulation d'images bitmap. Nous nous en tiendrons ici à des pièces rectangulaires afin de nous concentrer avant tout sur le code.

Configurer la classe

L'animation du puzzle classique est tout à fait identique à celle du puzzle coulissant. Elle contient trois images et la seconde appelle `startJigsawPuzzle`. Les mêmes instructions `import` sont en outre requises :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.net.URLRequest;
    import flash.geom.*;
    import flash.utils.Timer;
```

Les variables `numPiecesHoriz` et `numPiecesVert` sont listées tout en haut de la classe. Ces constantes ont toutes les chances d'être modifiées parce que vous pouvez faire varier le nombre de pièces d'un puzzle classique pour créer différents niveaux de difficulté. Nous allons créer un puzzle de 8 × 6 pour cet exemple :

```
    public class JigsawPuzzle extends MovieClip {
        // Nombre de pièces
        const numPiecesHoriz:int = 8;
        const numPiecesVert:int = 6;
```

La largeur et la hauteur des pièces seront décidées après que l'image aura été importée et que le programme aura pu en connaître la taille :

```
        // Taille des pièces
        var pieceWidth:Number;
        var pieceHeight:Number;
```

Comme pour le puzzle coulissant, nous stockons toutes les pièces du puzzle sous forme d'objets dans le tableau `puzzleObjects` :

```
// Pièces du jeu
var puzzleObjects:Array;
```

C'est à ce point que le jeu de puzzle classique prend un autre chemin que celui du puzzle coulissant. Au lieu de placer les pièces du puzzle directement sur la scène, nous les plaçons dans deux sprites. Le sprite `selectedPieces` contiendra toutes les pièces de puzzle qui sont actuellement déplacées. Le sprite `otherPieces` contiendra tout le reste.

La technique qui consiste à placer des groupes d'objets d'affichage à l'intérieur de sprites est un excellent moyen de regrouper des objets similaires. Comme vous le verrez dans la suite de cet exemple, vous pouvez utiliser `addChild` pour déplacer un objet d'affichage d'un sprite à un autre.

```
// Deux niveaux de sprite
var selectedPieces:Sprite;
var otherPieces:Sprite;
```

Lorsque le joueur sélectionne une pièce à faire glisser, il se peut qu'il sélectionne une unique pièce ou un groupe de pièces reliées. Au lieu d'une unique variable pointant vers la pièce déplacée, nous avons donc besoin d'un tableau pour stocker une ou plusieurs pièces :

```
// Pièces déplacées
var beingDragged:Array  = new Array();
```

La fonction constructeur du jeu, en plus d'appeler `loadBitmap`, crée les deux sprites dont nous avons besoin et les ajoute à la scène. L'ordre dans lequel ces sprites sont ajoutés est important parce que nous souhaitons que le sprite `selectedPieces` se trouve au-dessus du sprite `otherPieces` :

```
// Chargement de l'image et configuration des sprites
public function startJigsawPuzzle() {
    // Chargement de l'image bitmap
    loadBitmap("jigsawimage.jpg");

    // Configuration des deux sprites
    otherPieces = new Sprite();
    selectedPieces = new Sprite();
    addChild(otherPieces);
    addChild(selectedPieces); // selected on top
}
```

Chargement et découpage de l'image

L'image est chargée de la même manière que pour le puzzle coulissant. Je ne reviendrai ainsi pas sur la fonction `loadBitmap` parce qu'elle est identique à la précédente.

Chargement de l'image bitmap

La fonction `loadingDone` n'est pas si éloignée de l'ancienne non plus. Une fois l'image importée et les valeurs `pieceWidth` et `pieceHeight` calculées, nous appelons `makePuzzlePieces` pour découper l'image.

Il y a une différence importante concernant notre manière de procéder ici avec les calculs de `pieceWidth` et `pieceHeight`. La fonction `Math.floor` est utilisée avec une division par dix pour restreindre les nombres de la largeur et de la hauteur à des multiples de dix. Ainsi, si nous choisissions sept pièces dans la largeur pour 400 pixels, nous aurions 57,14 pixels pour chaque pièce. Nous utiliserons cependant une grille de 10 × 10 pour permettre aux joueurs de caler côte à côte les pièces plus aisément. En arrondissant cela à 50, nous nous assurons que les largeurs et hauteurs s'aligneront sur une grille de 10 × 10. Nous reviendrons sur ce point lorsque nous construirons la fonction `lockPieceToGrid` par la suite.

Pour finir, nous ajoutons deux écouteurs événementiels. Le premier est un événement `ENTER_FRAME` pour le mouvement des pièces durant leur déplacement. Le second est un événement `MOUSE_UP` sur la scène. Il sert à obtenir les événements de relâchement de souris qui signalent la fin du déplacement.

L'utilisateur clique sur une pièce pour commencer à la faire glisser et l'événement `MOUSE_DOWN` agit sur la pièce elle-même. Pourtant, ensuite, lorsque le déplacement est terminé, nous ne pouvons pas compter sur le fait que la souris se trouve toujours sur la même pièce ni même sur une pièce quelconque. Il est possible que le joueur déplace le curseur rapidement et que celui-ci se trouve légèrement à l'écart des pièces. Les événements de souris sont reconduits vers la scène ; il est donc plus sûr de placer un écouteur `MOUSE_UP` sur la scène afin de s'assurer que nous en soyons notifiés.

```
// Fin du chargement de l'image bitmap, découpage
private function loadingDone(event:Event):void {
    // Créer nouvelle image pour contenir les données bitmap chargées
    var image:Bitmap = Bitmap(event.target.loader.content);
    pieceWidth = Math.floor((image.width/numPiecesHoriz)/10)*10;
    pieceHeight = Math.floor((image.height/numPiecesVert)/10)*10;

    // Placer les données bitmap chargées dans l'image
    var bitmapData:BitmapData = image.bitmapData;

    // Découper en pièces de puzzle
    makePuzzlePieces(bitmapData);

    // Configurer les événements de mouvement et de relâchement de souris
    addEventListener(Event.ENTER_FRAME,movePieces);
    stage.addEventListener(MouseEvent.MOUSE_UP,liftMouseUp);
}
```

Découper les pièces du puzzle

Le principe de base pour la découpe des pièces est identique dans ce jeu. Nous n'aurons cependant pas besoin de définir d'emplacement pour les pièces parce qu'elles seront disposées de manière aléatoire un peu plus tard.

Une fois que les sprites sont créés, ils sont ajoutés à `otherPieces`, le sprite du dessous parmi les deux sprites que nous avons créés.

Les éléments `puzzleObject` sont aussi légèrement différents. Au lieu de `currentLoc` et `homeLoc`, nous n'aurons que `loc`, un objet `Point` qui nous indique à quel endroit appartient la pièce de puzzle dans le puzzle complet. `0,0` correspond par exemple à la pièce supérieure gauche.

Nous avons aussi ajouté une propriété `dragOffset` aux pièces du puzzle. Nous l'utilisons pour positionner chaque pièce à la distance appropriée du curseur pendant le déplacement :

```
// Découper l'image bitmap en pièces
private function makePuzzlePieces(bitmapData:BitmapData) {
    puzzleObjects = new Array();
    for(var x:uint=0;x<numPiecesHoriz;x++) {
        for (var y:uint=0;y<numPiecesVert;y++) {
            // Créer la nouvelle image bitmap de pièce de puzzle et le sprite
            var newPuzzlePieceBitmap:Bitmap = new Bitmap(new BitmapData(pieceWidth,
            pieceHeight));
            newPuzzlePieceBitmap.bitmapData.copyPixels(bitmapData,new Rectangle
            (x*pieceWidth,y*pieceHeight,pieceWidth,pieceHeight),new Point(0,0));
            var newPuzzlePiece:Sprite = new Sprite();
            newPuzzlePiece.addChild(newPuzzlePieceBitmap);

            // Placer dans le sprite du bas
            otherPieces.addChild(newPuzzlePiece);

            // Créer des objets à stocker dans le tableau
            var newPuzzleObject:Object = new Object();
            newPuzzleObject.loc = new Point(x,y); // location in puzzle
            newPuzzleObject.dragOffset = null; // offset from cursor
            newPuzzleObject.piece = newPuzzlePiece;
            newPuzzlePiece.addEventListener(MouseEvent.MOUSE_DOWN,clickPuzzlePiece);
            puzzleObjects.push(newPuzzleObject);
        }
    }

    // Rendre les emplacements des pièces aléatoires
    shufflePieces();
}
```

La fonction `shuffle` sélectionne des emplacements aléatoires pour toutes les pièces du puzzle. Nous ne nous préoccupons pas de savoir si les pièces se retrouvent les unes sur les autres ni comment elles sont réparties. Le but est que les pièces se retrouvent comme si elles étaient tombées de la boîte. La Figure 6.6 présente un exemple de cette répartition aléatoire.

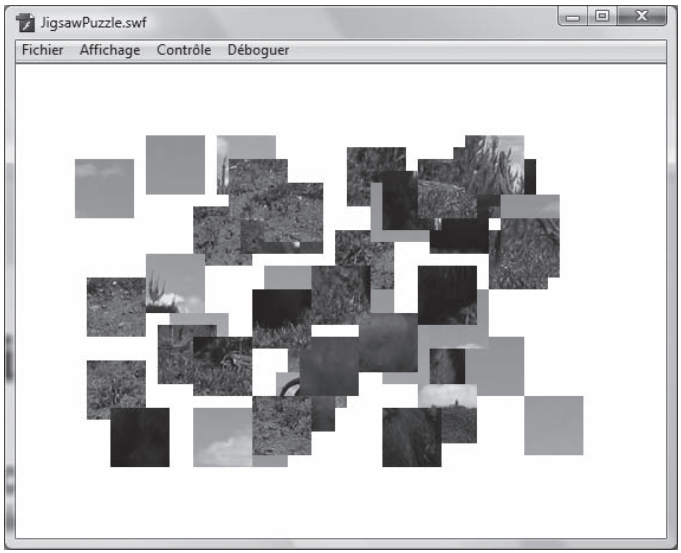


Figure 6.6
Les pièces du puzzle sont aléatoirement disposées à l'écran.

```
// Emplacements aléatoires pour les pièces
public function shufflePieces() {
    // Sélectionner x et y aléatoires
    for(var i in puzzleObjects) {
        puzzleObjects[i].piece.x = Math.random()*400+50;
        puzzleObjects[i].piece.y = Math.random()*250+50;
    }
    // Verrouiller toutes les pièces sur une grille 10x10
    lockPiecesToGrid();
}
```

La dernière ligne de `shufflePieces` appelle `lockPieceToGrid`. Cette fonction parcourt en boucle toutes les pièces du puzzle et les déplace vers l'emplacement le plus proche d'une grille de 10 × 10. Par exemple, si une pièce se trouve à `43,87`, elle est déplacée à `40,90`.

`lockPieceToGrid` est utilisée parce qu'il s'agit d'un moyen simple de permettre au joueur de rapprocher une pièce d'une autre sans avoir à l'aligner au pixel près pour que les pièces se "collent" l'une à l'autre. Normalement, si une pièce se trouve ne serait-ce qu'à un pixel d'une autre, elle ne s'y colle pas.

En plaçant toutes les pièces sur une grille de 10 × 10, nous les amenons à se coller parfaitement entre elles dès qu'elles se trouvent à moins de dix pixels les unes des autres.

```
// Prendre toutes les pièces et les aligner sur la grille 10x10
public function lockPiecesToGrid() {
    for(var i in puzzleObjects) {
        puzzleObjects[i].piece.x =
10*Math.round(puzzleObjects[i].piece.x/10);
        puzzleObjects[i].piece.y =
10*Math.round(puzzleObjects[i].piece.y/10);
    }
}
```

Comme nous utilisons des pièces de puzzle qui sont des multiples de dix en largeur et en hauteur, il est préférable d'utiliser une image source dont les dimensions sont elles-mêmes des multiples de dix. Une image de 400 × 300, par exemple, serait utilisée complètement. Une image de 284 × 192 perdra en revanche une partie à droite et en bas pour pouvoir se limiter à des pièces dont les dimensions sont des multiples de dix.

Faire glisser les pièces du puzzle

Lorsqu'un joueur clique sur une pièce du puzzle, plusieurs choses doivent se passer avant que la pièce ne puisse se déplacer avec le curseur. La première consiste à déterminer sur quelle pièce l'utilisateur a cliqué.

Trouver la pièce sur laquelle le joueur a cliqué

Cette opération s'effectue en parcourant en boucle les `puzzleObjects` jusqu'à trouver l'un d'entre eux dont la propriété `piece` correspond à `event.currentTarget`.

Cette pièce est ensuite ajoutée à un tableau `beingDragged` vide. La valeur `dragOffset` de cette pièce est en outre calculée en fonction de la distance entre l'emplacement du clic et celui du sprite.

Le sprite est déplacé du sprite `otherPieces` du dessous vers le sprite `selectedPieces` du dessus. Un appel à `addChild` suffit à cela. Dès lors que le joueur fait glisser la pièce du puzzle, celle-ci flotte au-dessus des autres, qui restent en dessous dans le sprite `otherPieces` :

```
public function clickPuzzlePiece(event:MouseEvent) {
    // Emplacement du clic
    var clickLoc:Point = new Point(event.stageX, event.stageY);

    beingDragged = new Array();
```

```
        // Trouver la pièce sur laquelle l'utilisateur a cliqué
        for(var i in puzzleObjects) {
            if (puzzleObjects[i].piece == event.currentTarget) { // this is it
                // Ajouter à la liste des pièces déplacées
                beingDragged.push(puzzleObjects[i]);
                // Calculer le décalage du curseur
                puzzleObjects[i].dragOffset = new Point(clickLoc.x -
                puzzleObjects[i].piece.x, clickLoc.y -
                puzzleObjects[i].piece.y);
                // Déplacer du sprite du dessous à celui du dessus
                selectedPieces.addChild(puzzleObjects[i].piece);
                // Trouver les autres pièces collées à celle-ci
                findLockedPieces(i,clickLoc);
                break;
            }
        }
    }
```

Trouver les pièces liées

Le plus intéressant lorsque le joueur clique sur une pièce est l'appel à `findLockedPieces`. C'est que la pièce sur laquelle le joueur a cliqué n'est pas forcément isolée. Elle peut être déjà jointe à d'autres pièces du puzzle suite aux précédents déplacements. Toutes les pièces jointes à celle sur laquelle l'utilisateur a cliqué doivent également être ajoutées à la liste `beingDragged`.

Pour déterminer si une pièce est collée à une autre, il faut suivre une série d'étapes. Ce processus commence en créant une liste ordonnée de toutes les pièces à l'exception de celle sur laquelle l'utilisateur a cliqué. Cette liste est triée en fonction de la distance par rapport à la pièce cliquée.

La commande `sortOn` est un puissant outil pour trier des listes d'objets. À supposer que le tableau ne contienne que des objets similaires possédant chacun la même propriété de tri, vous pouvez rapidement et aisément trier la liste. Par exemple, le tableau `[{a: 4, b: 7}, {a: 3, b:12}, {a: 9, b: 17}]` peut être trié sur `a` en utilisant simplement `myArray.sortOn("a");`.

Le code qui suit crée un tableau avec les propriétés `dist` et `num` pour chacun de ses éléments. La première correspond à la distance entre la pièce et celle sur laquelle l'utilisateur a cliqué. La seconde correspond au numéro de la pièce dans `puzzleObjects` :

```
// Trouver les pièces qui doivent se déplacer ensemble
public function findLockedPieces(clickedPiece:uint, clickLoc:Point) {
    // Obtenir la liste des objets puzzle triée par la distance à l'objet cliqué
    var sortedObjects:Array = new Array();
```

```
        for (var i in puzzleObjects) {
            if (i == clickedPiece) continue;
            sortedObjects.push(
    {dist: Point.distance(puzzleObjects[clickedPiece].loc,puzzleObjects[i].loc), num: i});
        }
        sortedObjects.sortOn("dist" ,Array.DESCENDING);
```

Maintenant que nous avons un tableau de pièces trié, nous pouvons le parcourir en boucle et vérifier chaque pièce afin de voir si elle est liée à la pièce sélectionnée.

Pour commencer, nous vérifions la position `x` et `y` de la pièce. Nous souhaitons voir si la pièce est correctement positionnée par rapport à la sélection. Par exemple, si les pièces font 50 de large et 50 de haut et si la pièce originale se trouve à `170,240`, la pièce directement à gauche doit se trouver à `120,240`. La pièce un cran plus loin à gauche doit être à `70,240`, et ainsi de suite.

La Figure 6.7 présente quelques exemples de pièces qui se relient et ne se relient pas.

À la Figure 6.7, l'exemple du haut présente deux pièces proches l'une de l'autre, mais pas suffisamment pour se relier. L'exemple suivant présente des pièces parfaitement positionnées, mais entre lesquelles il manque une pièce de liaison. La pièce à gauche se trouve exactement à la bonne distance de celle de droite, mais les pièces ne se relient pas parce qu'elles ne se trouvent pas côte à côte et que la pièce qui doit les unir est manquante.

Le troisième exemple présente deux pièces reliées. Elles sont adjacentes et à distance parfaite. Le quatrième est similaire, avec cette fois trois pièces reliées.

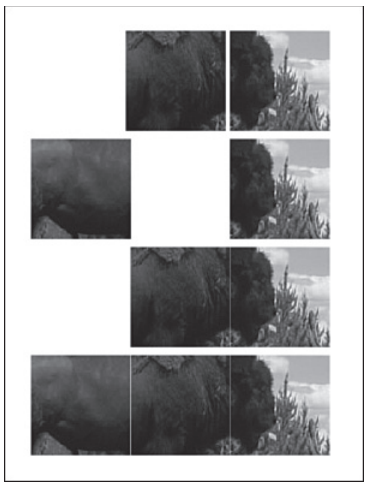


Figure 6.7
Les deux exemples du haut ne se relient pas, contrairement aux deux du bas.

La première étape consiste donc à voir si la pièce examinée est correctement placée. La seconde étape consiste à voir si elle est reliée aux pièces trouvées jusque-là. Ce travail est délégué à la fonction `isConnected`, que nous allons examiner dans un instant.

Si la pièce est effectivement reliée, nous pouvons l'ajouter à la liste `beingDragged`, positionner sa propriété `dragOffset` et l'ajouter au sprite `selectedPieces`.

Tout cela peut être placé dans une boucle `do` afin de pouvoir répéter le processus plusieurs fois. Il est possible d'avoir un ensemble de pièces liées sous la forme d'un U et que l'utilisateur ait souhaité saisir ce groupe par l'une des extrémités du U. Cela signifierait que l'autre bout du U ne serait pas reconnu comme une pièce liée. En revanche, si nous parcourons en boucle les pièces non liées à nouveau, le U sera trouvé. Nous pouvons donc définir une variable `Boolean` appelée `oneLineFound` et la positionner à `true` lorsque nous trouvons un lien. Si nous parcourons toutes les pièces non liées sans trouver de lien, il faut en conclure que nous les avons toutes. Sans cela, nous continuons à effectuer la boucle :

```
        do {
            var oneLinkFound:Boolean = false;
            // Examiner chaque objet en commençant par le plus proche
            for(i=sortedObjects.length-1;i>=0;i--) {
                var n:uint = sortedObjects[i].num; // actual object number
                // Calculer la position relative à l'objet cliqué
                var diffX:int = puzzleObjects[n].loc.x - puzzleObjects[clickedPiece].loc.x;
                var diffY:int = puzzleObjects[n].loc.y - puzzleObjects[clickedPiece].loc.y;
                // Voir si cet objet est correctement placé
    // pour être lié à celui cliqué
                if (puzzleObjects[n].piece.x ==
    (puzzleObjects[clickedPiece].piece.x + pieceWidth*diffX)) {
                    if (puzzleObjects[n].piece.y == (puzzleObjects[clickedPiece].piece.y +
                                                     pieceHeight*diffY)) {
                        // Voir si cet objet est adjacent à un objet déjà sélectionné
                        if (isConnected(puzzleObjects[n])) {
                            // Ajouter à la liste de sélection et définir décalage
                            beingDragged.push(puzzleObjects[n]);
                            puzzleObjects[n].dragOffset = new Point(clickLoc.x -
                                                          puzzleObjects[n].piece.x, clickLoc.y-
                                                          puzzleObjects[n].piece.y);
                            // Passer vers le sprite du haut
                            selectedPieces.addChild(puzzleObjects[n].piece);
                            // Lien trouvé, supprimer du tableau
                            oneLinkFound = true;
                            sortedObjects.splice(i,1);
                        }
                    }
                }
            }
        } while (oneLinkFound);
    }
```

La clé de la fonction `isConnected` tient au fait que nous avons déjà trié la liste de pièces en fonction de leur distance vis-à-vis de la première. C'est important, car nous devons progresser en nous écartant de la pièce de départ pour notre recherche de nouvelles pièces reliées. À mesure que nous examinons chaque nouvelle pièce, les pièces entre la pièce considérée et la pièce de départ auront selon toute probabilité déjà été examinées et ajoutées au tableau `beingDragged`. Cela minimise le nombre de fois où nous aurons à parcourir le tableau des pièces.

Par exemple, si l'utilisateur a cliqué sur la pièce `2,0` et que nous examinions ensuite `0,0`, nous pouvons en déduire qu'elle n'est pas reliée parce que `1,0` ne fait pas partie de `beingDragged`. Ensuite, nous examinons `1,0` et constatons qu'elle est reliée, mais il est trop tard pour ajouter `0,0` car nous l'avons déjà examinée. Nous devons donc examiner `1,0` en premier parce qu'elle est plus proche, puis `0,0`, qui est plus éloignée.

Si vous pensez que ce processus est compliqué, considérez qu'il serait généralement effectué en utilisant un processus de calcul appelé récursivité. On parle de récursivité lorsqu'une fonction s'appelle elle-même. C'est aussi un casse-tête pour bien des débutants en informatique. J'ai donc créé les méthodes de ce chapitre afin d'éviter spécifiquement la récursivité.

Déterminer si les pièces sont reliées

La fonction `isConnected` récupère la différence entre la position horizontale et verticale de la pièce examinée et de chaque pièce déjà dans `beingDragged`. Si elle constate qu'elle se trouve à un cran de distance horizontalement ou verticalement (mais pas les deux à la fois), elle est effectivement reliée :

```
// Prend un objet et détermine s'il est directement à côté de celui déjà sélectionné
public function isConnected(newPuzzleObject:Object):Boolean {
    for(var i in beingDragged) {
        var horizDist:int = Math.abs(newPuzzleObject.loc.x - beingDragged[i].loc.x);
        var vertDist:int = Math.abs(newPuzzleObject.loc.y - beingDragged[i].loc.y);
        if ((horizDist == 1) && (vertDist == 0)) return true;
        if ((horizDist == 0) && (vertDist == 1)) return true;
    }
    return false;
}
```

Déplacer les pièces

Nous savons enfin quelles pièces doivent être déplacées. Elles se trouvent maintenant toutes soigneusement dans `beingDragged`, qui est utilisé par `movePieces` dans chaque image pour les repositionner toutes :

```
// Déplacer toutes les pièces sélectionnées en fonction de l'emplacement de la souris
public function movePieces(event:Event) {
```

```
        for (var i in beingDragged) {
            beingDragged[i].piece.x = mouseX - beingDragged[i].dragOffset.x;
            beingDragged[i].piece.y = mouseY - beingDragged[i].dragOffset.y;
        }
    }
```

Terminer le mouvement

Lorsque le joueur relâche la souris, le glissement s'interrompt. Nous devons déplacer toutes les pièces du sprite `selectedPieces` vers le sprite `otherPieces`. Nous appellerons aussi `lockPiecesToGrid` afin de nous assurer qu'elles peuvent être mises en correspondance avec les pièces qui n'ont pas été déplacées.

Lorsque la fonction `addChild` est utilisée pour ramener les pièces vers le sprite `otherPieces`, les pièces sont replacées au-dessus de toutes les autres pièces à cet endroit. Ce principe fonctionne bien parce que nous flottons déjà au-dessus. L'effet de superposition est ainsi conservé.

```
    // La scène envoie événement de relâchement de souris, fin du déplacement
    public function liftMouseUp(event:MouseEvent) {
        // Caler toutes les pièces sur la grille
        lockPiecesToGrid();
        // Ramener les pièces dans le sprite du dessous
        for(var i in beingDragged) {
            otherPieces.addChild(beingDragged[i].piece);
        }
        // Nettoyer le tableau des pièces déplacées
        beingDragged = new Array();

        // Voir si la partie est terminée
        if (puzzleTogether()) {
            cleanUpJigsaw();
            gotoAndStop("gameover");
        }
    }
```

Fin de partie

Lorsque la souris est relâchée, nous devons également vérifier si la partie est terminée. Pour cela, nous pouvons parcourir en boucle toutes les pièces du puzzle et comparer leurs positions avec celle de la première pièce en haut à gauche. Si toutes se trouvent dans la position correcte par rapport à cette première, nous savons que le puzzle est terminé :

```
public function puzzleTogether():Boolean {
    for(var i:uint=1;i<puzzleObjects.length;i++) {
        // Obtenir la position relative au premier objet
        var diffX:int = puzzleObjects[i].loc.x - puzzleObjects[0].loc.x;
        var diffY:int = puzzleObjects[i].loc.y - puzzleObjects[0].loc.y;
        // Voir si cet objet est correctement placé
// Pour être relié au premier
        if (puzzleObjects[i].piece.x !=
        (puzzleObjects[0].piece.x + pieceWidth*diffX)) return false;
        if (puzzleObjects[i].piece.y !=
        (puzzleObjects[0].piece.y + pieceHeight*diffY)) return false;
    }
    return true;
}
```

La fonction de nettoyage obligatoire tire parti de notre système à deux sprites. Nous pouvons tout simplement les supprimer de la scène et positionner les variables qui les référencent à `null`. Nous devons aussi positionner `puzzleObjects` et `beginDragged` à `null`, ainsi que supprimer les événements `ENTER_FRAME` et `MOUSE_UP` :

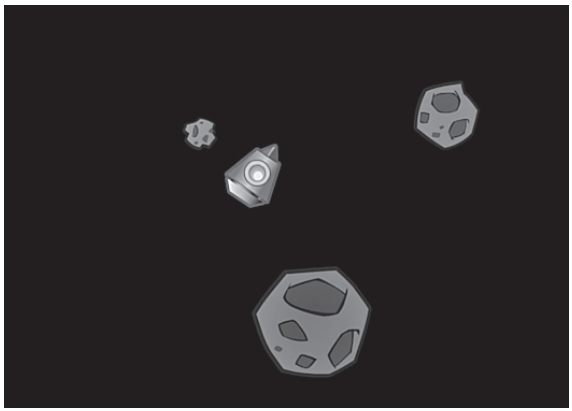
```
public function cleanUpJigsaw() {
    removeChild(selectedPieces);
    removeChild(otherPieces);
    selectedPieces = null;
    otherPieces = null;
    puzzleObjects = null;
    beingDragged = null;
    removeEventListener(Event.ENTER_FRAME,movePieces);
    stage.removeEventListener(MouseEvent.MOUSE_UP,liftMouseUp);
}
```

Modifier le jeu

Les développeurs de jeu ont trouvé de nombreux moyens de rendre les puzzles sur ordinateur plus intéressants que leurs équivalents physiques.

Si vous vous êtes déjà amusé à créer des filtres bitmap en ActionScript, l'occasion peut être trouvée d'exploiter cette technique ici. L'ajout de néons, d'ombres portées ou de biseaux aux pièces peut réellement contribuer à leur donner de l'allure.

Vous pouvez également ajouter une rotation des pièces afin de compliquer le puzzle. Les pièces peuvent être pivotées de 90, 180 ou 270 degrés et doivent être ramenées à 0 pour créer le puzzle. Vous devez évidemment permettre au joueur de faire pivoter les pièces après les avoir reliées, ce qui requiert du code assez complexe pour pouvoir faire pivoter en bloc les pièces collées les unes aux autres. Ne vous risquez dans une telle aventure que si vous avez atteint le grade de ceinture noire en programmation ActionScript.



7

Direction et mouvement : astéroïdes

Au sommaire de ce chapitre :

- Utiliser Math pour faire pivoter et déplacer des objets
- Air Raid II
- Space Rocks

Les jeux présentés au Chapitre 5 faisaient appel à un mouvement horizontal et vertical simple. Le déplacement le long de l'axe horizontal ou vertical est très facile à programmer. Malheureusement, les jeux d'arcade se révèlent plus exigeants.

Un grand nombre de jeux requièrent que vous donniez la possibilité au joueur de tourner et de se déplacer. Par exemple, les jeux de conduite proposent à la fois un mouvement de volant et un mouvement vers l'avant. Les jeux dans l'espace le requièrent également et peuvent même permettre au joueur de tirer des projectiles dans la direction où pointe le vaisseau.

Utiliser *Math* pour faire pivoter et déplacer des objets

Codes sources

http://flashgameu.com

A3GPU07_RotationMath.zip

La combinaison de la rotation et du mouvement requiert des calculs plus complexes que les simples opérations d'addition, de soustraction, de multiplication et de division. Il faut utiliser des fonctions trigonométriques fondamentales comme le sinus, le cosinus et l'arctangente.

Si vous n'êtes pas fort en mathématiques, n'ayez crainte : ActionScript va prendre en charge l'essentiel du boulot.

Les fonctions *Sin* et *Cos*

Au Chapitre 5, nous avons utilisé des variables comme `dx` et `dy` pour définir la différence des positions horizontales et verticales. Un objet dont la variable `dx` valait 5 et la variable `dy` valait 0 se déplaçait de 5 pixels vers la droite et de 0 vers le haut ou le bas.

Comment en revanche déterminer les valeurs de `dx` et `dy` si nous ne connaissons que l'angle de rotation d'un objet ? Supposons que le joueur ait la possibilité de faire tourner un objet, comme une voiture, dans n'importe quelle direction. Il fait pointer sa voiture légèrement vers le bas et la droite. Ensuite, il avance. Vous devez dans ce cas modifier les propriétés `x` et `y` de la voiture, mais vous ne connaissez que l'angle de rotation de la voiture.

La propriété `rotation` de n'importe quel objet d'affichage correspond à un nombre compris entre –180 et 180, c'est-à-dire au nombre de degrés de l'objet par rapport à son degré de rotation 0 d'origine. Vous pouvez modifier la rotation exactement comme les valeurs d'emplacement `x` et `y`. La rotation peut aussi être plus précise, par exemple en spécifiant un angle de 23,76 degrés. Si vous souhaitez tourner lentement, vous pouvez donc ajouter 0,01 à cette propriété à chaque image ou à chaque période temporelle.

C'est ici qu'entrent en jeu les fonctions `Math.cos` et `Math.sin`. Elles permettent de calculer `dx` et `dy` en n'utilisant qu'un angle.

La Figure 7.1 présente la logique mathématique de `Math.cos` et `Math.sin`. Il s'agit d'un graphique de cercle. `Math.cos` et `Math.sin` nous permettent de trouver n'importe quel point sur le cercle en fonction d'un angle donné.

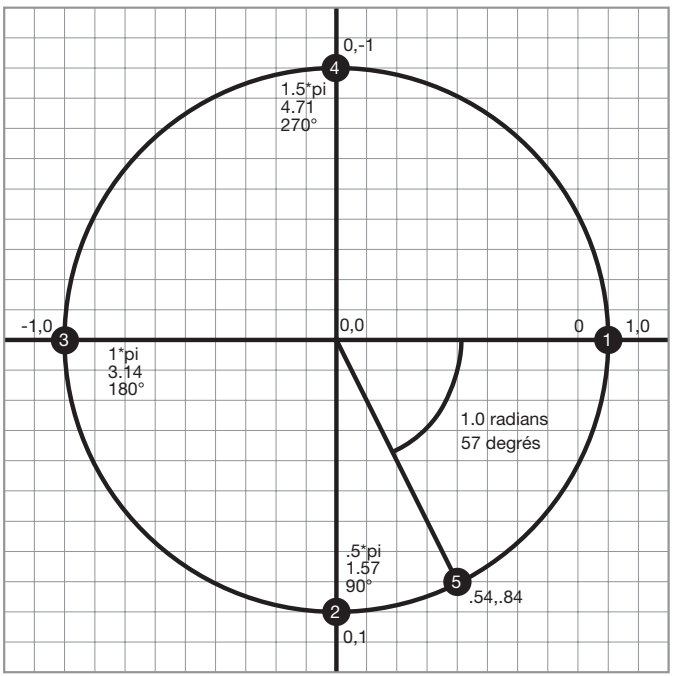


Figure 7.1
Ce graphique d'un cercle montre la relation entre un angle et l'emplacement x et y d'un point sur le cercle.

Si l'angle en question est 0, `Math.cos` et `Math.sin` retournent respectivement `1.0` et `0.0`. Nous obtenons ainsi le point numéro 1, dont la valeur de `x` est `1.0` et la valeur de `y` est `0.0`. Un objet ayant pivoté de 0 degré se déplace donc du centre du cercle vers le point 1.

Si l'objet pointe à 90 degrés, `Math.cos` et `Math.sin` retournent respectivement `0.0` et `1.0`. Il s'agit du point numéro 2. Un objet pivoté de 90 degrés se déplace donc droit vers le bas.

Vous pouvez voir de la même manière où conduisent les angles de 180 et 270 degrés : le premier tout droit à gauche et le second directement vers le haut.

La Figure 7.1 présente les radians sous forme de multiples de pi, les radians bruts et les degrés. Les radians et les degrés sont deux moyens différents de mesurer les angles. Un cercle complet fait 360 degrés, ce qui correspond à 2 × pi radians. Pi vaut approximativement 3,14, de sorte que 360 degrés = 6,26 radians.

ActionScript utilise à la fois des degrés et des radians. Les degrés sont utilisés par la propriété `rotation` des objets. Les radians sont utilisés par les fonctions mathématiques comme `Math.cos` et `Math.sin`. Nous effectuerons donc constamment la conversion d'une unité de mesure vers l'autre.

Ces quatre directions sont faciles à déterminer sans utiliser `Math.cos` et `Math.sin`. C'est cependant pour les angles intermédiaires que nous dépendons véritablement de ces fonctions trigonométriques.

Le cinquième point se trouve à un angle d'environ 57 degrés. On peut difficilement se passer cette fois de `Math.cos` et `Math.sin` pour déterminer à quel emplacement du cercle il correspond. Le résultat est `0.54` dans la direction `x` et `0.84` dans la direction `y`. Si un objet devait se déplacer d'un pixel vers l'avant tout en pointant à 57 degrés, nous nous retrouverions donc au niveau du point 5.

Il est important de comprendre que ces 5 points (comme tous les autres points sur le cercle) se trouvent à une distance exactement identique du centre. Le problème pour atteindre ces points concerne donc non pas la vitesse de l'objet mais la direction empruntée.
L'autre point important à retenir est que `Math.cos` et `Math.sin` retournent toujours des valeurs comprises entre –1,0 et 1,0. Elles partent du principe que le cercle fait 1,0 unité en rayon. Ainsi, si un objet se trouve à 57 degrés et se déplace de 1,0 unité, il atteint `0.54`, `0.84`. Si sa vitesse est `5`, nous multiplions en revanche ce résultat par 5 et obtenons `2.70`, `4.20`.

Utiliser *Cos* et *Sin* pour piloter une voiture

Considérons un exemple simple pour expliquer l'utilisation de ces fonctions trigonométriques. Les animations **MovingCar.fla** et **MovingCar.as** proposent un jeu de simulation de conduite élémentaire. Une voiture est placée au milieu de l'écran et le joueur peut utiliser les touches fléchées de gauche et de droite pour tourner ainsi que la touche fléchée du haut pour avancer. La Figure 7.2 présente la voiture à l'écran.

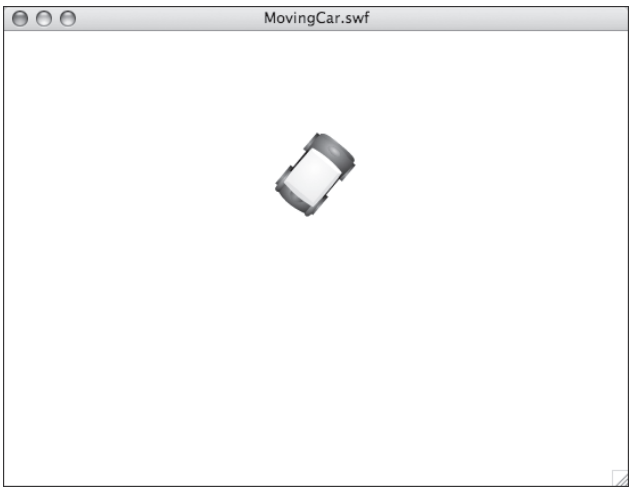


Figure 7.2
Cet exemple simple de simulation de conduite permet au joueur de tourner et d'avancer.

Nous utiliserons du code analogue à celui du jeu Air Raid du Chapitre 5. Nous ferons appel à trois variables booléennes, `leftArrow`, `rightArrow` et `upArrow`. Toutes ces variables seront positionnées à `true` lorsque le joueur appuiera sur la touche correspondante et à `false` lorsqu'il la relâchera.

Utiliser *Math* pour faire pivoter et déplacer des objets

Voici le début de la classe, avec les écouteurs et le code qui gère les touches fléchées. Vous remarquerez que nous n'avons besoin d'aucune instruction import supplémentaires pour utiliser les fonctions `Math`. Celles-ci font partie de la bibliothèque ActionScript standard :

```
package {
    import flash.display.*;
    import flash.events.*;

    public class MovingCar extends MovieClip {
        private var leftArrow, rightArrow, upArrow: Boolean;

        public function MovingCar() {

            // Déplacer à chaque image
            addEventListener(Event.ENTER_FRAME, moveCar);

            // Réagir aux événements de touche
            stage.addEventListener(KeyboardEvent.KEY_DOWN,keyPressedDown);
            stage.addEventListener(KeyboardEvent.KEY_UP,keyPressedUp);
        }

        // Positionner les variables de touche à true
        public function keyPressedDown(event:KeyboardEvent) {
            if (event.keyCode == 37) {
                leftArrow = true;
            } else if (event.keyCode == 39) {
                rightArrow = true;
            } else if (event.keyCode == 38) {
                upArrow = true;
            }
        }
```

```
// Positionner les variables de touche à false
public function keyPressedUp(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = false;
    } else if (event.keyCode == 39) {
        rightArrow = false;
    } else if (event.keyCode == 38) {
        upArrow = false;
    }
}
```

La fonction `moveCar` est appelée à chaque image. Elle examine chacune des valeurs booléennes et détermine si l'une d'entre elles vaut `true`. Dans le cas des touches fléchées de gauche et de droite, la propriété `rotation` du clip `car` est modifiée et la voiture est pivotée.

Vous remarquerez que nous n'utilisons pas une animation temporelle cette fois. Vous influerez donc sur la vitesse de rotation et de déplacement si vous modifiez la cadence d'images de l'animation.

Si la touche fléchée du haut est enfoncée, nous appelons la fonction `moveForward` :

```
// Tourner ou avancer la voiture
public function moveCar(event:Event) {
    if (leftArrow) {
        car.rotation -= 5;
    }
    if (rightArrow) {
        car.rotation += 5;
    }
    if (upArrow) {
        moveForward();
    }
}
```

Voilà où nous utilisons nos calculs mathématiques. Si la touche fléchée du haut est enfoncée, nous calculons d'abord l'angle en radians de la voiture. Nous connaissons la rotation de la voiture, mais elle s'exprime en degrés. Pour convertir les degrés en radians, nous divisons notre résultat par 360 (le nombre de degrés dans un cercle) puis multiplions deux fois par pi (le nombre de radians dans un cercle).

Nous utiliserons souvent cette conversion. Il vaut donc la peine de la décomposer pour plus de clarté :

1. Diviser par 360 pour convertir la valeur comprise entre 0 et 360 en une valeur comprise entre 0 et 1,0.
2. Multiplier par 2 × pi pour convertir la valeur comprise entre 0 et 1,0 en une valeur comprise entre 0 et 6,28.

```
radians = 2 * pi * (degrees / 360)
```

À l'inverse, lorsque nous souhaitons effectuer la conversion des radians en degrés, nous procédons de la manière suivante :

1. Diviser par 2 × pi pour convertir la valeur comprise entre 0 et 6,28 en une valeur comprise entre 0 et 1,0.
2. Multiplier par 360 pour convertir la valeur comprise entre 0 et 1,0 en une valeur comprise entre 0 et 360.

```
degrees = 360 * radians / (2 * pi)
```

Comme les degrés et les radians mesurent des angles, les valeurs se répètent elles-mêmes tous les 360 degrés ou 2 × pi radians. Ainsi, 0 degré équivaut à 360 degrés, de même que 90 et 450. Il en va de même avec les valeurs négatives. Par exemple, 270 degrés équivalent à –90 degrés. En fait, la propriété `rotation` de n'importe quel objet retourne toujours une valeur comprise entre –180 et 180, qui équivaut à pi et -pi radians.

Maintenant que nous avons l'angle en radians, nous le fournissons à `Math.cos` et `Math.sin` pour obtenir les valeurs `dx` et `dy` pour le mouvement. Nous multiplions en outre le tout par `speed`, une valeur que nous avons définie précédemment dans la fonction. Cette action déplace la voiture de 5 pixels par image au lieu de 1 pixel par image.

Pour finir, nous changeons les propriétés `x` et `y` de la voiture afin de la déplacer réellement :

```
        // Claculer vitesse x et y et déplacer la voiture
        public function moveForward() {
            var speed:Number = 5.0;
            var angle:Number = 2*Math.PI*(car.rotation/360);
            var dx:Number = speed*Math.cos(angle);
            var dy:Number = speed*Math.sin(angle);
            car.x += dx;
            car.y += dy;
        }
    }
}
```

Jouez avec l'animation **MovingCar.fla**. Faites tourner la voiture à différents angles et appuyez sur la touche fléchée du haut pour la voir avancer. Représentez-vous les fonctions `Math.cos` et `Math.sin` qui traduisent l'angle en une quantité horizontale et verticale de mouvement.

Ensuite, amusez-vous. Appuyez sur les touches fléchées de gauche et du haut en même temps pour faire tourner la voiture en cercle. Cela revient à tourner le volant de votre voiture en même temps que vous accélérez. La voiture continue à tourner.

Si l'on oublie l'accélération un instant, on obtient une amusante petite simulation de conduite. Au Chapitre 12, nous créerons une véritable simulation de conduite complexe au cœur duquel mécanisme figureront toujours ces opérations avec `Math.cos` et `Math.sin`.

Calculer un angle à partir d'un emplacement

Si `Math.sin` et `Math.cos` vous permettent d'obtenir les coordonnées `x` et `y` d'un angle, nous devrons aussi occasionnellement obtenir un angle à partir d'un ensemble de coordonnées `x` et `y`. Pour cela, nous devons utiliser un calcul d'arctangente. La fonction ActionScript pour cela est `Math.atan2`.

La Figure 7.3 montre comment la fonction `arctangent` fonctionne. Le Point 1 est situé à 6,5 sur la grille. Pour trouver son angle, nous prenons la distance `y` et la distance `x` et les fournissons à `Math.atan2`. Le résultat sera 0,69 radian, soit environ 40 degrés.

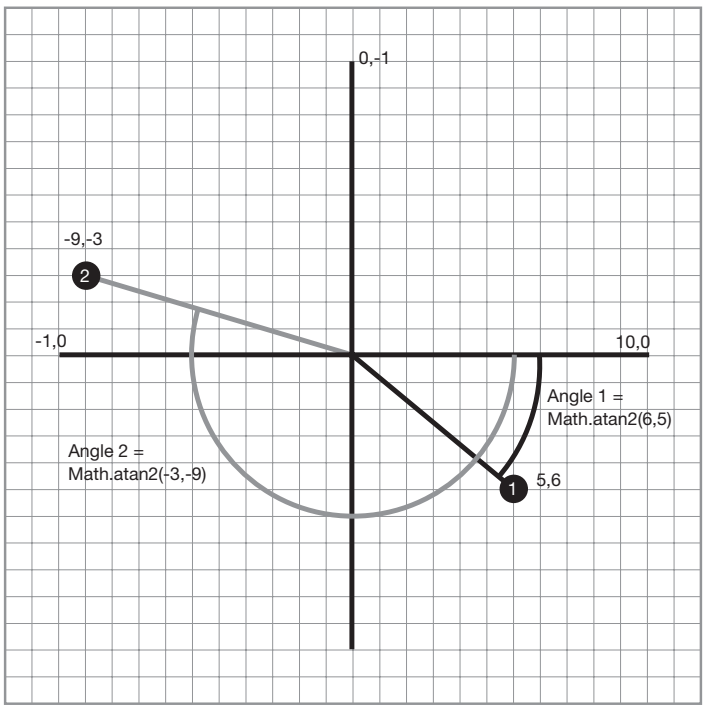


Figure 7.3
Les angles de ces deux points peuvent être déterminés en utilisant `Math.atan2`.

Le second point se trouve à –9,–3. En fournissant ces valeurs à `Math.atan2`, nous obtenons –2,82 radians, soit –162 degrés (qui équivaut à 198 degrés). La fonction `Math.atan2` produit un résultat toujours compris entre –180 et 180.

Il existe aussi une fonction `Math.atan`. Elle prend un paramètre : le rapport entre la distance `y` et la distance `x`. Vous l'utiliseriez donc de la manière suivante : `Math.atan(dy/dx)`. Il s'agit de la fonction mathématique arctangente classique. Le problème est que vous ne savez pas si le résultat est en avançant ou en reculant. Par exemple, –5/3 équivaut à 5/–3. L'un correspond à 121 degrés, l'autre, à –60 degrés. La fonction `Math.atan` retourne –60 degrés dans les deux cas. La fonction `Math.atan2` fournit à la différence l'angle approprié.

Créons un exemple simple en utilisant une flèche. Vous le trouverez dans les fichiers source **PointingArrow.fla** et **PointingArrow.as**.

La flèche est située au centre de l'écran (emplacement 275,200). Examinez la Figure 7.4. Vous remarquerez que le point d'alignement du clip se trouve au centre de la flèche. Lorsque vous faites pivoter un clip, il tourne autour de ce point. Vous remarquerez en outre que la flèche pointe directement vers la droite. Une rotation de 0 correspond à cette direction. Tout objet créé dans le seul but de pivoter doit donc être créé en étant ainsi tourné vers la droite.

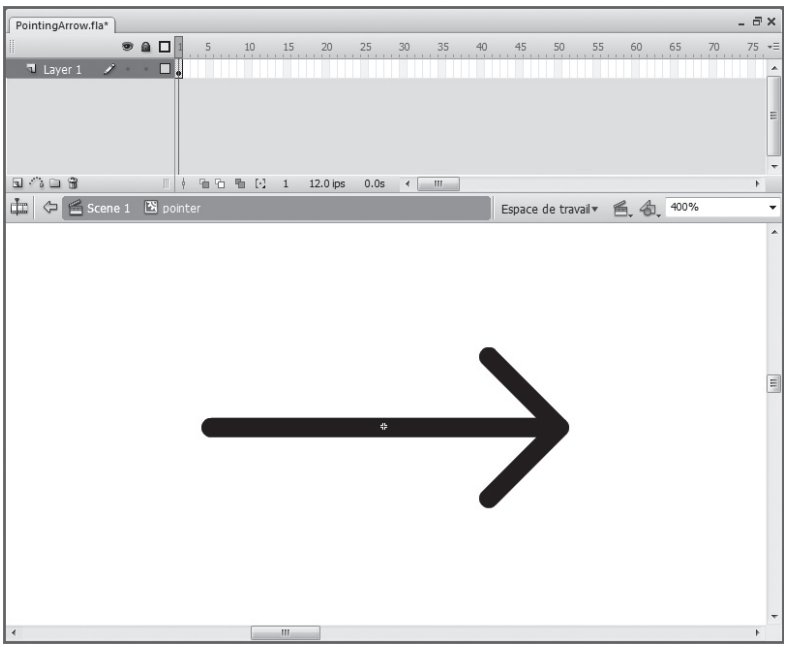


Figure 7.4
Il est plus simple de faire pivoter des objets tournés vers la droite, avec le centre du clip au centre de la rotation.

Cette flèche pointe "vers" le curseur. Nous avons donc un point d'origine pour la flèche à 275,200 et un point de destination de l'emplacement du curseur. Comme il est facile de déplacer le curseur et de changer `mouseX` et `mouseY`, il s'agit d'un moyen rapide d'expérimenter `Math.atan2`.

La classe concise suivante provenant de **PointingArrow.as** appelle une fonction à chaque image. Cette fonction calcule les valeurs `dx` et `dy` à partir des distances entre le curseur et l'emplacement de la flèche. Elle utilise ensuite `Math.atan2` pour calculer l'angle en radians. Elle le convertit en degrés et attribue le résultat à la propriété `rotation` de la flèche :

```
package {
    import flash.display.*;
    import flash.events.*;

    public class PointingArrow extends MovieClip {

        public function PointingArrow() {
            addEventListener(Event.ENTER_FRAME, pointAtCursor);
        }

        public function pointAtCursor(event:Event) {
            // Récupérer l'emplacement relatif de la souris
            var dx:Number = mouseX - pointer.x;
            var dy:Number = mouseY - pointer.y;

            // Déterminer l'angle, le convertir en degrés
            var cursorAngle:Number = Math.atan2(dy,dx);
            var cursorDegrees:Number = 360*(cursorAngle/(2*Math.PI));

            // Pointer vers le curseur
            pointer.rotation = cursorDegrees;
        }
    }
}
```

Lorsque vous exécutez cette animation, la flèche pointe à tout moment vers le curseur (voir Figure 7.5). À tout le moins, `mouseX` et `mouseY` se mettent à jour, ce qui ne se produit que lorsque le curseur survole l'animation Flash.

Maintenant que vous savez comment utiliser la trigonométrie pour observer et contrôler les positions et le mouvement des objets, nous pouvons les appliquer à des jeux.

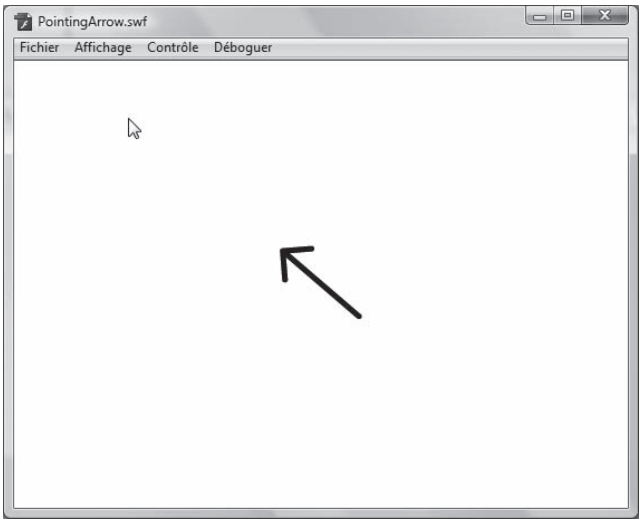


Figure 7.5
La flèche pointe vers le curseur lorsque celui-ci survole l'animation.

Combinés l'un à l'autre, ces deux exemples simples peuvent produire d'intéressants résultats. Par exemple, que se passe-t-il si la voiture devait être guidée par la position de la souris vis-à-vis de la voiture ? Elle pointerait vers la souris et lorsque vous vous déplaceriez, s'avancerait vers la souris à tout moment. Autrement dit, elle pourchasserait la souris. Que se passerait-il donc si le joueur devait piloter une voiture comme dans le premier exemple et qu'une seconde voiture pointait vers la souris et avançait par elle-même ? La seconde voiture viendrait pourchasser la première ! Le site ***http://flashgameu.com*** *en propose un exemple.*

Air Raid II

Codes sources
http://flashgameu.com
A3GPU07_AirRaid2.zip

Dans le jeu Air Raid du Chapitre 5, vous déplaciez un canon dans un sens et dans l'autre en utilisant les touches fléchées. Ce mouvement vous permettait de cibler différentes parties du ciel alors que vous tiriez vers le haut.

Grâce à `Math.sin` et `Math.cos`, nous pouvons maintenant modifier ce jeu de manière à immobiliser le canon tout en lui permettant de modifier son angle pour atteindre différentes cibles.

Modifier le canon

La première chose à faire est de modifier le clip `AAGun` afin d'avoir des canons pivotants. Nous allons séparer complètement du clip la base de la tourelle et la placer dans son propre clip, `AAGunBase`. Les canons resteront dans `AAGun`, mais nous les recentrerons afin que le point de pivot se trouve au centre et que les canons pointent vers la droite (voir Figure 7.6).

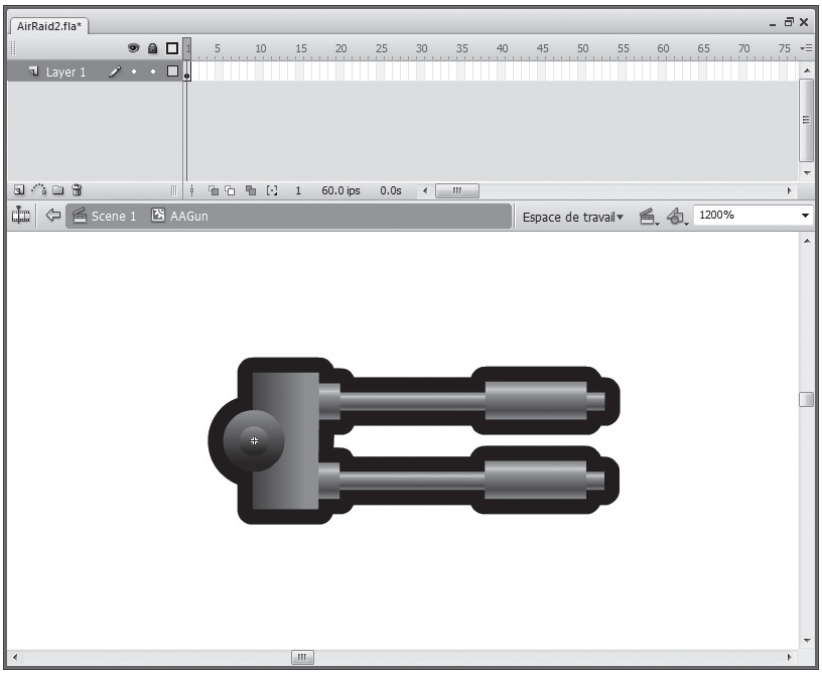


Figure 7.6
Les canons doivent pointer vers la droite pour correspondre aux valeurs cos et sin.

L'idée est de modifier le moins possible le jeu Air Raid d'origine. Nous allons prendre les mêmes valeurs pour les appuis sur les touches fléchées et les utiliser pour modifier la propriété `rotation` de `AAGun`, au lieu de la valeur `y`.

Vous pourriez aussi utiliser un autre jeu de touches pour définir la rotation (par exemple, A et S ou la touche Commande et le point). Ensuite, utilisez les touches fléchées pour déplacer le canon : vous aurez alors une arme qui se déplace latéralement et dont les canons peuvent pivoter.

Nous définissons encore les valeurs `x` et `y` de la mitraillette, mais également la valeur `rotation` en la fixant à –90. Cette valeur de –90 signifie que la mitraillette commence en pointant vers le haut. Nous

restreindrons la valeur de la rotation tout comme nous avons restreint le mouvement horizontal dans la première version du jeu Air Raid. Cette fois, les valeurs seront comprises entre –170 et–20 degrés, ce qui correspond à un angle de 50 degrés vers la gauche ou la droite de la verticale.

Voici donc notre nouveau code **AAGun.as**. Recherchez les lignes dans le code suivant qui impliquent la variable `newRotation` et la propriété `rotation` :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class AAGun extends MovieClip {
        static const speed:Number = 150.0;
        private var lastTime:int; // Temps de l'animation

        public function AAGun() {
            // Emplacement initial de la mitraillette
            this.x = 275;
            this.y = 340;
            this.rotation = -90;

            // Mouvement
            addEventListener(Event.ENTER_FRAME,moveGun);
        }

        public function moveGun(event:Event) {
            // Calculer le temps écoulé
            var timePassed:int = getTimer()-lastTime;
            lastTime += timePassed;

            // Position actuelle
            var newRotation = this.rotation;

            // Pivoter à gauche
            if (MovieClip(parent).leftArrow) {
                newRotation -= speed*timePassed/1000;
            }

            // Pivoter à droite
            if (MovieClip(parent).rightArrow) {
                newRotation += speed*timePassed/1000;
            }
```

```
            // Vérifier les limites
            if (newRotation < -170) newRotation = -170;
            if (newRotation > -20) newRotation = -20;

            // Repositionner
            this.rotation = newRotation;
        }
        // Supprimer de l'écran et supprimer les événements
        public function deleteGun() {
            parent.removeChild(this);
            removeEventListener(Event.ENTER_FRAME,moveGun);
        }
    }
}
```

Vous remarquerez que la valeur `speed` fixée à 150 reste la même. Il serait normalement fort probable que le passage du mouvement horizontal au mouvement de rotation requierre une adaptation de la valeur de vitesse mais, dans le cas présent, la valeur de 150 fonctionne dans les deux cas.

Changer les balles

La classe **Bullets.as** doit changer pour que les balles se déplacent vers le haut en suivant un angle au lieu d'être parfaitement verticales.

Le graphisme doit changer également. Les balles doivent pointer vers la droite et être centrées sur le point d'alignement. La Figure 7.7 présente le nouveau clip `Bullet`.

La classe doit changer afin d'ajouter les variables de mouvement `dx` et `dy`. Ces variables seront calculées à partir de l'angle auquel la balle a été tirée, qui est un nouveau paramètre passé dans la fonction `Bullet`.

En outre, la balle doit commencer à une certaine distance du centre de la mitraillette ; dans le cas présent, elle doit être à 40 pixels du centre. Les valeurs `Math.cos` et `Math.sin` sont toutes deux utilisées pour calculer la position d'origine de la balle et calculer les valeurs `dx` et `dy`.

En outre, la rotation du clip `Bullet` sera définie pour correspondre à la rotation de la mitraillette. Les balles commenceront donc juste au-dessus de la fin du canon, en pointant vers l'extérieur du canon, et continueront de se déplacer en s'éloignant avec le même angle :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;
```

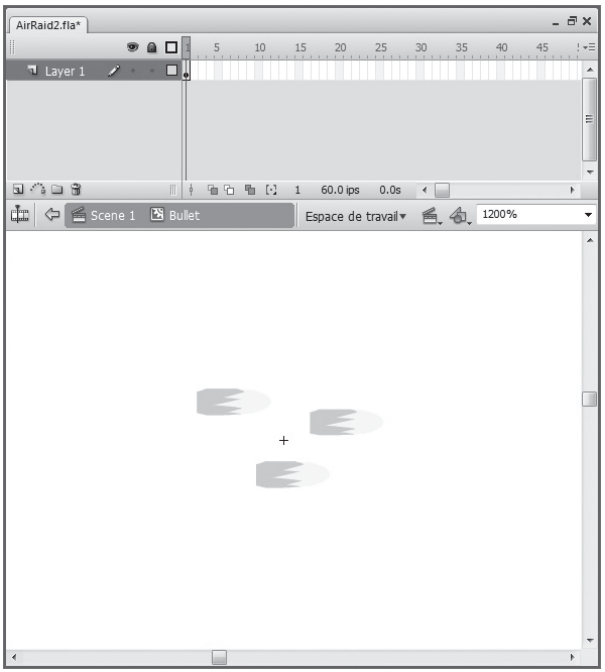


Figure 7.7
Le nouveau clip Bullet recentre le graphisme et le fait pointer vers la droite.

```
public class Bullet extends MovieClip {
    private var dx,dy:Number; // Vitesse
    private var lastTime:int;

    public function Bullet(x,y:Number, rot: Number, speed: Number) {
        // Définir position de départ
        var initialMove:Number = 35.0;
        this.x = x + initialMove*Math.cos(2*Math.PI*rot/360);
        this.y = y + initialMove*Math.sin(2*Math.PI*rot/360);
        this.rotation = rot;

        // Obtenir la vitesse
        dx = speed*Math.cos(2*Math.PI*rot/360);
        dy = speed*Math.sin(2*Math.PI*rot/360);

        // Configurer l'animation
        lastTime = getTimer();
        addEventListener(Event.ENTER_FRAME,moveBullet);
```

```
        }

        public function moveBullet(event:Event) {
            // Calculer le temps écoulé
            var timePassed:int = getTimer()-lastTime;
            lastTime += timePassed;

            // Déplacer la balle
            this.x += dx*timePassed/1000;
            this.y += dy*timePassed/1000;

            // La balle a dépassé le haut de l'écran
            if (this.y < 0) {
                deleteBullet();
            }
        }

        // Supprimer la balle de la scène et de la liste
        public function deleteBullet() {
            MovieClip(parent).removeBullet(this);
            parent.removeChild(this);
            removeEventListener(Event.ENTER_FRAME,moveBullet);
        }

    }
}
```

Changements apportés à AirRaid2.as

Des changements doivent être apportés à la classe principale afin de s'adapter aux nouvelles versions de `AAGun` et `Bullet`. Considérons chaque changement. Nous allons créer une nouvelle classe appelée **AirRaid2.as** et changer la classe du document de l'animation afin de pointer sur cette classe. N'oubliez pas aussi de changer la définition de classe en haut du code en remplaçant `AirRaid` par `AirRaid2`.

Dans les définitions de variable de la classe, nous devons ajouter le nouveau clip `AAGunBase` ainsi que conserver le clip `AAGun` :

```
private var aagun:AAGun;
private var aagunbase:AAGunBase;
```

Dans `startAirRaid`, nous devons tenir compte du fait que la mitraillette est maintenant représentée par deux clips. `AAGunBase` ne possède pas de classe à soi : nous devons faire correspondre sa position avec celle de `AAGun`.

Vous pouvez aussi supprimer complètement `AAGunBase` en utilisant un graphisme différent ou caler les canons sur un graphisme qui fait partie de l'arrière-plan.

```
// Créer la mitraillette
aagun = new AAGun();
addChild(aagun);
aagunbase = new AAGunBase();
addChild(aagunbase);
aagunbase.x = aagun.x;
aagunbase.y = aagun.y;
```

Le seul autre changement nécessaire doit être opéré dans la fonction `fireBullet`. Cette fonction doit passer la rotation de la mitraillette à la classe `Bullet`, afin qu'elle connaisse la direction dans laquelle tirer la balle. Nous ajouterons donc ce troisième paramètre de manière à le faire correspondre au troisième paramètre dans la fonction `Bullet` qui crée une nouvelle balle :

```
var b:Bullet = new Bullet(aagun.x,aagun.y,aagun.rotation,300);
```

Si nous créions ce jeu de toutes pièces, nous pourrions même ne pas inclure les deux premiers paramètres qui se réfèrent à la position de la mitraillette. Puisque celle-ci ne se déplace plus, elle restera en effet à la même position. Comme nous avions déjà du code qui s'occupait de relier le point de départ de la balle à la position de la mitraillette, nous pouvons le conserver ici et tirer parti du fait qu'un seul emplacement dans le code sert à déterminer la position de la mitraillette.

Nous sommes parvenus à changer la classe **AirRaid2.as**. En fait, si nous n'avions pas eu besoin pour des raisons esthétiques d'ajouter `AAGunBase` à l'animation, nous n'aurions eu que ce dernier changement à opérer dans **AirRaid2.as**. Ce point illustre bien la flexibilité d'ActionScript lorsque vous utilisez une classe différente pour chaque élément mouvant.

Nous avons maintenant un jeu Air Raid II entièrement transformé qui utilise une mitraillette pivotante mais stationnaire.

Space Rocks

Codes sources

http://flashgameu.com

A3GPU07_SpaceRocks.zip

L'un des jeux vidéo les plus classiques de tous les temps s'est appelé Asteroids. Ce jeu d'arcade vectoriel a été commercialisé par Atari en 1979. Ces graphismes se limitaient à de simples lignes unies, il émettait des sons plus que rudimentaires et l'on pouvait aisément tricher et gagner. Malgré cela, il se révélait tout à fait captivant.

Le joueur contrôlait un petit vaisseau. Vous pouviez tourner, tirer et voler dans l'écran. Face à vous se trouvaient quelques grands astéroïdes qui se déplaçaient à vitesse variable et dans des directions aléatoires. Vous pouviez les faire éclater en astéroïdes plus petits en tirant dessus. Les astéroïdes les plus petits disparaissaient lorsque l'on tirait dessus. Si un astéroïde vous touchait, vous perdiez une vie.

Nous allons construire un jeu qui fonctionne selon le même principe, avec un vaisseau, des astéroïdes et des missiles. Nous utiliserons même l'une des fonctionnalités les plus avancées du jeu original : un bouclier de protection.

Éléments du jeu et conception

Avant de commencer, nous devons décider à quoi ressemblera notre jeu Space Rocks. Nous n'aurons pas besoin de créer un document de conception complet, mais quelques listes nous aideront à rester concentrés à mesure que nous construisons le jeu de toutes pièces.

Les éléments du jeu sont un vaisseau, des astéroïdes et des missiles. Chacune des variantes est présentée à la Figure 7.8.

Examinons les capacités du vaisseau. Voici une liste de ce que le vaisseau peut faire :

- Il apparaît au milieu de l'écran au départ, en position stationnaire.
- Il tourne à gauche lorsque la touche fléchée de gauche est enfoncée.
- Il tourne à droite lorsque la touche fléchée de droite est enfoncée.
- Il accélère vers l'avant lorsque la touche fléchée du haut est enfoncée.
- Il se déplace en fonction de sa vélocité.
- Il génère un bouclier lorsque la touche Z est enfoncée.

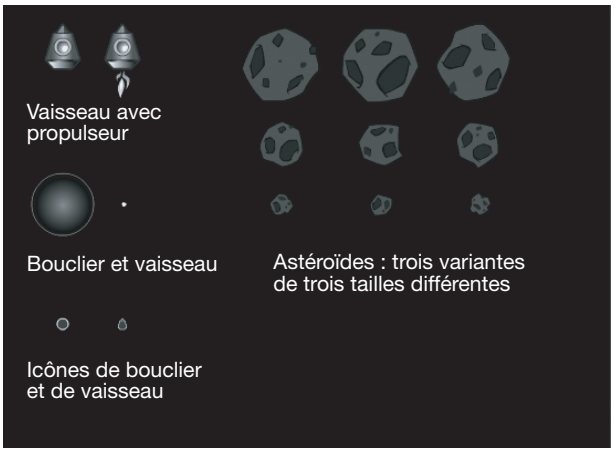


Figure 7.8
Les éléments du jeu pour Space Rocks.

Le vaisseau lance un missile. Voici ce que font les missiles :

- Ils sont créés lorsque le joueur appuie sur la barre d'espace.
- Leur vélocité et leur position sont déterminées par l'emplacement et la rotation du vaisseau.
- Ils se déplacent en fonction de leur vélocité.

Voici maintenant ce que font les astéroïdes :

- Ils ont une vélocité et une vitesse de départ aléatoires.
- Ils se déplacent en fonction de leur vélocité.
- Ils pivotent avec une vitesse de rotation définie.
- Ils possèdent trois tailles différentes : grande, moyenne et petite.

Les collisions font l'essentiel de ce jeu. Il en existe deux types : celle du missile avec un astéroïde et celle d'un astéroïde avec le vaisseau.

Lorsqu'un missile et un astéroïde entrent en collision, l'astéroïde d'origine est supprimé. S'il s'agissait d'un astéroïde version "grand", deux astéroïdes de taille moyenne apparaissent au même emplacement. S'il s'agissait d'une version "moyen", deux "petits" astéroïdes apparaissent au même emplacement. Les petits astéroïdes se contentent de disparaître, sans être remplacés par aucun. Le missile est également supprimé après la collision.

Lorsqu'un astéroïde entre en collision avec le vaisseau, il se comporte comme si un missile l'avait touché. Le vaisseau est pour sa part supprimé. Le joueur a trois vies. S'il ne s'agit pas de la dernière vie du joueur, celui-ci obtient un autre vaisseau qui apparaît au centre de l'écran au bout de deux secondes.

Si le joueur détruit tous les astéroïdes et qu'il n'en reste plus à l'écran, le niveau est terminé. Après un court délai, une nouvelle vague d'astéroïdes apparaît, un peu plus rapide que la précédente.

Dans la plupart des versions des années 1970 du jeu Asteroids, il existait une vitesse maximale limite des astéroïdes. Cette contrainte permettait aux joueurs experts de jouer indéfiniment ou à tout le moins jusqu'à ce que le patron de la salle de jeux les en chasse ou que maman leur rappelle très fermement qu'il était l'heure de dîner.

Parmi les autres actions possibles du joueur, l'une consiste à générer un bouclier. L'appui sur la touche Z crée un bouclier autour du vaisseau pendant 3 secondes. Ce bouclier permet au vaisseau de traverser des astéroïdes sans se détruire. Les joueurs n'ont cependant que trois boucliers par vie et doivent les utiliser avec précaution.

L'un des aspects importants du jeu tient à ce que le vaisseau et les astéroïdes bouclent leur parcours autour de l'écran lorsqu'ils se déplacent. Si l'un d'entre eux quitte l'écran à gauche, il réapparaît à droite. S'il le quitte par le bas, il réapparaît en haut. Les missiles, pour leur part, vont jusqu'au bord de l'écran puis disparaissent.

Définir les graphismes

Il nous faut un vaisseau, des astéroïdes et un missile pour créer ce jeu. Le vaisseau est l'élément le plus complexe. Il doit avoir un état normal, un état avec un propulseur activé et une animation d'explosion lorsqu'il est touché. Il a aussi besoin d'un bouclier qui le recouvre de temps à autre.

La Figure 7.9 présente un clip du vaisseau qui explose. Plusieurs images sont utilisées. La première correspond au vaisseau sans les gaz et la seconde, au vaisseau avec ses gaz activés. Le reste des images consiste en une courte animation d'explosion.

Les boucliers correspondent en réalité à un autre clip placé à l'intérieur du clip du vaisseau. Ce clip est présent à la fois sur la première (sans gaz) et la deuxième (avec gaz) image. Nous désactiverons le bouclier en attribuant la valeur `false` à sa propriété `visible`. Lorsque nous en aurons besoin, nous le réactiverons en attribuant cette fois la valeur `true` à la propriété `visible`.

Les astéroïdes correspondront en fait à une série de clips. Il en existera trois pour les trois tailles : `Rock_Big`, `Rock_Medium` et `Rock_Small`. Les trois clips auront à leur tour trois images représentant les variantes des astéroïdes. Nous éviterons ainsi que tous les astéroïdes se ressemblent. La Figure 7.10 présente le clip `Rock_Big` dans lequel vous remarquerez les images-clés qui contiennent les trois variantes dans le scénario.

Le missile est l'élément le plus simple. Il s'agit simplement d'un petit point jaune. Deux autres clips sont aussi utilisés : `ShipIcon` et `ShieldIcon`. Il s'agit de versions miniatures du vaisseau et du bouclier. Nous les utiliserons pour afficher le nombre de vaisseaux et de boucliers restants.

Figure 7.9
Dans cette image du vaisseau, le bouclier et les gaz sont tous deux activés.

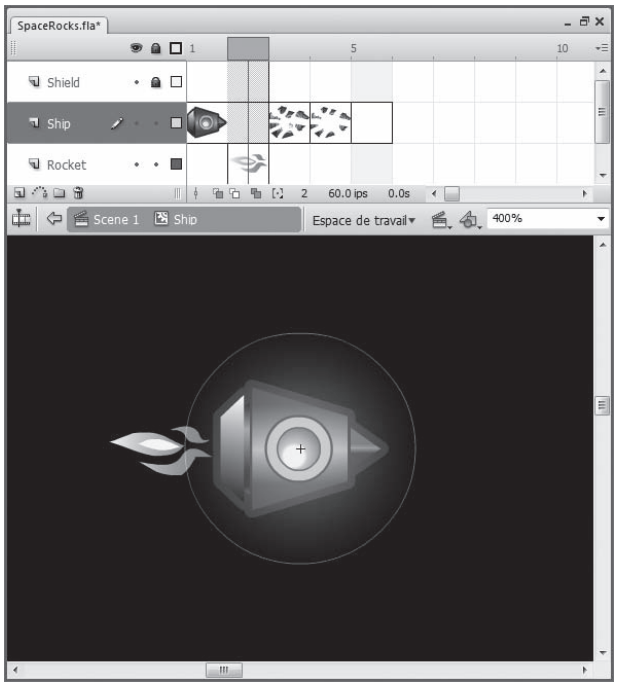
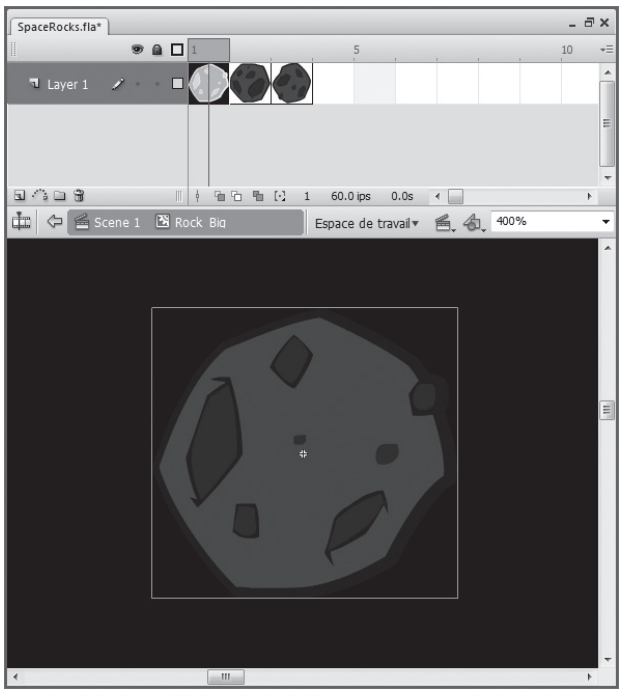


Figure 7.10
Chaque clip des astéroïdes inclut trois variantes de la même taille.



Le scénario principal est configuré de la manière habituelle : trois images, avec celle du milieu qui appelle `startSpaceRocks`. Il nous suffit maintenant de créer le code ActionScript pour donner vie au jeu.

Configurer la classe

Nous allons placer tout le code dans un fichier de classe **SpaceRocks.as**. Il s'agira du plus long fichier de classe que nous ayons créé jusque-là. L'avantage lié au fait d'utiliser un unique fichier de classe tient à ce que tout le code se trouve conservé dans un endroit. L'inconvénient tient à ce qu'il peut devenir long et difficile à manier.

Pour simplifier les choses, nous allons décomposer le code en plus petites sections, chacune traitant un élément de l'écran différent. Pour commencer, examinons cependant la déclaration de classe.

La classe requiert les importations habituelles pour gérer l'ensemble des objets et structures :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
    import flash.utils.getTimer;
    import flash.utils.Timer;
    import flash.geom.Point;
```

Un certain nombre de constantes permettent d'ajuster le comportement et la difficulté du jeu. Les vitesses sont toutes mesurées avec des unités calées sur les millisecondes, de sorte que la vitesse de rotation du vaisseau (`shipRotationSpeed`) est une vitesse plutôt rapide de 0,1/1000 soit 100 degrés par seconde. Les missiles se déplacent à 200 pixels par seconde et les propulseurs accélèrent le vaisseau à 150 pixels par seconde par seconde.

La vitesse est mesurée en unité de distance par temps, par exemple en pixels par seconde. L'accélération désigne la variation de vitesse au cours du temps : le nombre de pixels par seconde dont la vitesse change par seconde. Voilà pourquoi il faut écrire : pixels par seconde par seconde.

La vitesse des astéroïdes dépend du niveau. Elle est de 0,03 plus 0,02 fois le niveau, soit 0,05 pour le premier niveau, 0,07 pour le second, etc.

Nous définissons aussi le rayon du vaisseau, qui est à peu près de forme ronde. Nous utiliserons ce rayon pour détecter les collisions au lieu de nous appuyer sur la fonction `hitTestObject` :

```
public class SpaceRocks extends MovieClip {
    static const shipRotationSpeed:Number = .1;
    static const rockSpeedStart:Number = .03;
    static const rockSpeedIncrease:Number = .02;
    static const missileSpeed:Number = .2;
    static const thrustPower:Number = .15;
    static const shipRadius:Number = 20;
    static const startingShips:uint = 3;
```

Après les constantes, nous devons définir une série de variables à configurer par la suite. Voici les variables qui contiennent des références au vaisseau, aux astéroïdes et aux missiles :

```
// Objets du jeu
private var ship:Ship;
private var rocks:Array;
private var missiles:Array;
```

Ensuite, nous avons un minuteur d'animation qui sera utilisé pour mettre au pas tous les mouvements :

```
// Minuteur d'animation
private var lastTime:uint;
```

Les touches fléchées de gauche, de droite et du haut seront consignées par les valeurs booléennes suivantes :

```
// Touches fléchées
private var rightArrow:Boolean = false;
private var leftArrow:Boolean = false;
private var upArrow:Boolean = false;
```

La vélocité du vaisseau sera décomposée en deux valeurs de vitesse :

```
// Vélocité du vaisseau
private var shipMoveX:Number;
private var shipMoveY:Number;
```

Nous avons deux minuteurs. L'un correspond au délai entre le moment où le joueur perd un vaisseau et celui où le suivant apparaît. Nous l'utiliserons aussi pour imposer un délai avant l'affichage du prochain ensemble d'astéroïdes lorsque tous les astéroïdes ont été détruits. L'autre correspond à la durée d'activation du bouclier :

```
// Minuteurs
private var delayTimer:Timer;
private var shieldTimer:Timer;
```

Une variable `gameMode` peut se voir attribuer les valeurs `"play"` ou `"delay"`. Lorsqu'elle vaut `"delay"`, nous n'écoutons pas les appuis sur les touches du joueur. Une autre variable booléenne nous indique si le bouclier est actif, auquel cas le joueur ne peut être touché par les astéroïdes :

```
// Mode du jeu
private var gameMode:String;
private var shieldOn:Boolean;
```

L'ensemble de variables suivant concerne les boucliers et les vaisseaux. Les deux premières consignent le nombre de vaisseaux et de boucliers. Les deux suivantes sont des tableaux contenant des références aux icônes affichées à l'écran qui convoient ces informations au joueur :

```
// Vaisseaux et boucliers
private var shipsLeft:uint;
private var shieldsLeft:uint;
private var shipIcons:Array;
private var shieldIcons:Array;
```

Le score est stocké dans `gameScore`. Il est affiché dans un champ texte que nous allons créer nommé `scoreDisplay`. La variable `gameLevel` tient le registre du nombre d'ensembles d'astéroïdes qui ont été détruits :

```
// Score et niveau
private var gameScore:Number;
private var scoreDisplay:TextField;
private var gameLevel:uint;
```

Pour finir, nous avons deux sprites dans lesquels nous placerons tous les éléments du jeu. Le premier est `gameObjects` et correspondra à notre sprite principal. Nous placerons cependant les icônes de vaisseau et de bouclier ainsi que le score dans le sprite `scoreObjects` afin de les conserver à part :

```
// Sprites
private var gameObjects:Sprite;
private var scoreObjects:Sprite;
```

Démarrer le jeu

La fonction constructeur commencera par configurer les sprites. Il est important que les instructions `addChild` apparaissent dans cet ordre afin que les icônes et le score restent au-dessus des éléments du jeu :

```
// Démarrer le jeu
public function startSpaceRocks() {
```

```
        // Configurer les sprites
        gameObjects = new Sprite();
        addChild(gameObjects);
        scoreObjects = new Sprite();
        addChild(scoreObjects);
```

`gameLevel` est positionné à `1` et `shipsLeft`, à `3` d'après les constantes définies précédemment. `gameScore` est fixé à zéro également. Ensuite, un appel à `createShipIcons` et à `createScoreDisplay` configure le tout.

Nous les examinerons sous peu :

```
        // Réinitialiser les objets de score
        gameLevel = 1;
        shipsLeft = startingShips;
        gameScore = 0;
        createShipIcons();
        createScoreDisplay();
```

Il nous faut trois écouteurs, comme pour les jeux Air Raid. L'un sera un appel de fonction d'image général, les deux autres concerneront les appuis sur les touches :

```
        // Configurer les écouteurs
        addEventListener(Event.ENTER_FRAME,moveGameObjects);
        stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

Pour lancer le jeu, nous positionnons `gameMode` à `"delay"` et `shieldOn` à `false`, puis créons un tableau pour stocker les missiles et appelons deux fonctions pour lancer le jeu. La première crée le premier ensemble d'astéroïdes et la seconde crée le premier vaisseau. Comme ces deux fonctions seront par la suite appelées par des minuteurs événementiels, nous devons inclure `null` en paramètre afin de remplir l'emplacement qu'utilisera la valeur du minuteur événementiel par la suite :

```
        // Démarrer
        gameMode = "delay";
        shieldOn = false;
        missiles = new Array();
        nextRockWave(null);
        newShip(null);
    }
```

Objets d'affichage du score et de l'état

Le premier grand groupe de fonctions a affaire au nombre de vaisseaux que possède le joueur, au nombre de boucliers restants et au score du joueur. Ces nombres s'affichent dans les trois coins de l'écran.

Le score apparaît sous forme de texte dans le coin supérieur droit. Le nombre de vaisseaux est indiqué en affichant de zéro à trois icônes dans le coin inférieur gauche. Le nombre de boucliers est indiqué en affichant de zéro à trois icônes de bouclier dans le coin inférieur droit. La Figure 7.11 présente le jeu au début lorsque les trois éléments sont présents.

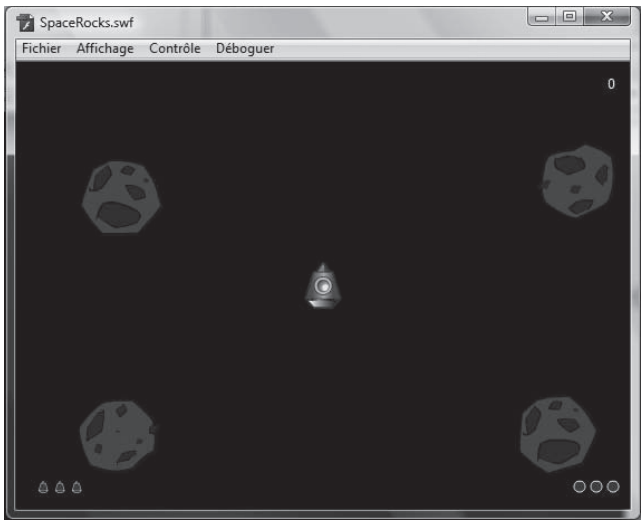


Figure 7.11
Le score est affiché dans le coin supérieur droit, le nombre de vies, en bas à gauche et le nombre de boucliers, en bas à droite.

Les deux fonctions suivantes s'occupent de créer les icônes de vaisseau et de bouclier en plaçant en boucle les trois éléments à l'écran. Ces éléments sont ajoutés à leurs tableaux respectifs afin que l'on puisse y faire référence et les supprimer par la suite :

```
// Dessiner le nombre de vaisseaux restants
public function createShipIcons() {
    shipIcons = new Array();
    for(var i:uint=0;i<shipsLeft;i++) {
        var newShip:ShipFs:Icon = new ShipIcon();
        newShip.x = 20+i*15;
        newShip.y = 375;
        scoreObjects.addChild(newShip);
        shipIcons. push(newShip);
    }
}
```

Voici une fonction similaire pour les icônes de bouclier :

```
// Dessiner le nombre de boucliers restants
public function createShieldIcons() {
    shieldIcons = new Array();
    for(var i:uint=0;i<shieldsLeft;i++) {
        var newShield:ShieldIcon = new ShieldIcon();
        newShield.x = 530-i*15;
        newShield.y = 375;
        scoreObjects.addChild(newShield);
        shieldIcons.push(newShield);
    }
}
```

Nous aurions aussi pu éviter d'utiliser des icônes et nous servir simplement de champs texte pour afficher le nombre de vaisseaux et de boucliers restants. Il faudrait moins de code, mais l'effet serait visuellement moins intéressant.

Pour l'affichage du score, nous devons créer un nouveau champ texte et définir ses propriétés. Nous utiliserons aussi une variable temporaire `TextFormat` pour définir le format texte par défaut (`defaultTextFormat`) du champ :

```
// Placer le score numérique en haut à droite
public function createScoreDisplay() {
    scoreDisplay = new TextField();
    scoreDisplay.x = 500;
    scoreDisplay.y = 10;
    scoreDisplay.width = 40;
    scoreDisplay.selectable = false;
    var scoreDisplayFormat = new TextFormat();
    scoreDisplayFormat.color = 0xFFFFFF;
    scoreDisplayFormat.font = "Arial";
    scoreDisplayFormat.align = "right";
    scoreDisplay.defaultTextFormat = scoreDisplayFormat;
    scoreObjects.addChild(scoreDisplay);
    updateScore();
}
```

À la fin de `createScoreDisplay`, nous appelons immédiatement `updateScore` pour placer un 0 dans le champ, car il s'agit de la valeur de `gameScore` à ce point. La fonction `updateScore` sera cependant utilisée plus tard aussi, à chaque fois que nous avons un changement dans le score :

```
// Nouveau score à afficher
public function updateScore() {
    scoreDisplay.text = String(gameScore);
}
```

Lorsqu'il est temps de supprimer un vaisseau ou un bouclier, nous devons dépiler un élément des tableaux `shipIcons` ou `shieldIcons` et utiliser `removeChild` sur `scoreObjects` pour effacer l'icône :

```
// Supprimer une icône de vaisseau
public function removeShipIcon() {
    scoreObjects.removeChild(shipIcons.pop());
}

// Supprimer une icône de bouclier
public function removeShieldIcon() {
    scoreObjects.removeChild(shieldIcons.pop());
}
```

Nous devons aussi ajouter des fonctions qui parcourent en boucle et suppriment toutes les icônes. Nous en avons besoin à la fin du jeu. Pour les boucliers, nous en avons besoin à la fin d'une vie. Nous souhaitons donner au joueur trois boucliers à chaque nouveau vaisseau. Nous supprimerons donc les icônes de bouclier et recommencerons lorsque cela se produira :

```
// Supprimer le reste des icônes de vaisseau
public function removeAllShipIcons() {
    while (shipIcons.length > 0) {
        removeShipIcon();
    }
}

// Supprimer le reste des icônes de bouclier
public function removeAllShieldIcons() {
    while (shieldIcons.length > 0) {
        removeShieldIcon();
    }
}
```

Mouvement du vaisseau et entrée du joueur

L'ensemble des fonctions suivantes concerne le vaisseau. La première fonction crée un nouveau vaisseau. Le reste des fonctions concerne son déplacement.

Créer un nouveau vaisseau

La fonction `newShip` est appelée au début du jeu mais également 2 secondes après la disparition d'un précédent vaisseau. Lors de ces occasions subséquentes, un minuteur se chargera de cet appel. Ce minuteur se verra passer un `TimerEvent`. Nous n'en aurons cependant pas besoin.

Lors des deuxième, troisième et quatrième fois où la fonction est appelée, le précédent vaisseau existe déjà. Il aura joué son animation d'explosion. À la fin de cette animation, une commande `stop` met en pause le clip à la dernière image qui est vide. Le vaisseau est donc toujours là, mais il est invisible. Nous vérifions que le vaisseau ne vaut pas `null` puis le supprimons et l'enlevons avant de poursuivre.

Dans d'autres jeux, il peut être souhaitable de supprimer le vaisseau dès que l'animation d'explosion est terminée. Dans ce cas, vous pouvez simplement placer un rappel à la classe principale à l'intérieur du scénario du vaisseau. Cet appel peut se trouver dans la dernière image de l'animation, afin d'indiquer que l'animation est terminée et que l'objet peut être supprimé.

```
// Créer un nouveau vaisseau
public function newShip(event:TimerEvent) {
    // Si le vaisseau existe, le supprimer
    if (ship != null) {
        gameObjects.removeChild(ship);
        ship = null;
    }
```

Ensuite, nous vérifions s'il reste des vaisseaux. S'il n'en reste pas, la partie est terminée :

```
    // Plus de vaisseau
    if (shipsLeft < 1) {
        endGame();
        return;
    }
```

Un nouveau vaisseau est créé, positionné et ramené à la première image qui contient un vaisseau normal gaz éteints. La `rotation` est fixée à `-90`, de manière à pointer vers le haut. Nous devons aussi supprimer le bouclier.

Ensuite, nous pouvons ajouter le clip au sprite `gameObjects` :

```
// Créer, positionner et ajouter le nouveau vaisseau
ship = new Ship();
ship.gotoAndStop(1);
ship.x = 275;
ship.y = 200;
ship.rotation = -90;
ship.shield.visible = false;
gameObjects.addChild(ship);
```

La vélocité du vaisseau est stockée dans les variables `shipMoveX` et `shipMoveY`. Maintenant que nous avons créé un vaisseau, `gameMode` peut passer de `"delay"` à `"play"` :

```
// Configuration des propriétés du vaisseau
shipMoveX = 0.0;
shipMoveY = 0.0;
gameMode = "play";
```

Pour chaque nouveau vaisseau, nous réinitialisons le nombre de boucliers à 3. Ensuite, nous devons dessiner les trois petites icônes de bouclier en bas de l'écran :

```
// Mettre en place les boucliers
shieldsLeft = 3;
createShieldIcons();
```

Lorsque le joueur perd un vaisseau et qu'un nouveau vaisseau apparaît, il existe une possibilité qu'il réapparaisse au milieu de l'écran exactement au moment où un astéroïde passe à cet endroit. Pour éviter que cela ne pose un problème, nous pouvons utiliser les boucliers. En activant ces derniers, le joueur est assuré de ne pas être victime d'une collision pendant 3 secondes.

Les jeux d'arcade de ce type évitaient ce problème à l'origine en attendant simplement que le milieu de l'écran devienne relativement vide avant de créer un nouveau vaisseau. Vous pourriez procéder de la même manière, en vérifiant la distance entre chaque astéroïde et le nouveau vaisseau et en imposant un délai supplémentaire de 2 secondes quand un projectile est trop proche.

Notez que nous ne souhaitons procéder ainsi que si ce n'est pas la première fois que le vaisseau apparaît. Au début du jeu, les astéroïdes feront aussi leur première apparition et se trouveront à des emplacements prédéfinis situés à l'écart du centre.

Lorsque nous appelons ici `startShield`, nous passons la valeur `true` afin d'indiquer qu'il s'agit d'un bouclier "gratuit". Il n'est pas décompté dans la réserve des trois boucliers du joueur :

```
        // Toutes les vies sauf la première commencent par un bouclier gratuit
        if (shipsLeft != startingShips) {
            startShield(true);
        }
    }
```

Gérer l'entrée clavier

Les deux fonctions suivantes s'occupent des appuis sur les touches. Comme avec Air Raid, nous surveillons les touches fléchées de gauche et de droite. Nous nous occupons aussi de la touche fléchée du haut. Nous réagissons également à la barre d'espace et à la touche Z.

Dans le cas de la touche fléchée du haut, nous activons le propulseur et allumons les gaz en demandant au vaisseau de passer à la seconde image qui fait apparaître des flammes à l'arrière.

La barre d'espace appelle `newMissile` et la touche Z, `startShield` :

```
    // Enregistrer les appuis sur les touches
    public function keyDownFunction(event:KeyboardEvent) {
        if (event.keyCode == 37) {
            leftArrow = true;
        } else if (event.keyCode == 39) {
            rightArrow = true;
        } else if (event.keyCode == 38) {
            upArrow = true;
            // Allumer les gaz
            if (gameMode == "play") ship.gotoAndStop(2);
        } else if (event.keyCode == 32) { // Barre d'espace
            newMissile();
        } else if (event.keyCode == 90) { // z
            startShield(false);
        }
    }
```

La fonction `keyUpFunction` coupe les gaz lorsque le joueur relâche la touche fléchée du haut :

```
    // Enregistrer les relâchements de touches
    public function keyUpFunction(event:KeyboardEvent) {
        if (event.keyCode == 37) {
            leftArrow = false;
```

```
        } else if (event.keyCode == 39) {
            rightArrow = false;
        } else if (event.keyCode == 38) {
            upArrow = false;
            // Éteindre les gaz
            if (gameMode == "play") ship.gotoAndStop(1);
        }
    }
```

Mouvement du vaisseau

Toutes les fonctions d'animation de ce jeu prennent `timeDiff` en paramètre. Cette variable équivaut à la variable `timePassed` des autres jeux avec animation mais, au lieu que chaque fonction d'animation calcule son propre temps écoulé, nous le calculons dans une unique fonction appelée `moveGameObjects` qui appelle ensuite les trois fonctions d'animation et leur passe cette valeur. Tous les objets se déplacent alors en parfaite synchronisation les uns avec les autres.

Le mouvement du vaisseau peut concerner son pivotement, son déplacement vers l'avant ou les deux. Si les touches fléchées de gauche ou de droite sont enfoncées, le vaisseau tourne en fonction de `timeDiff` et de la constante `shipRotationSpeed`.

Si la touche fléchée du haut est enfoncée, le vaisseau s'accélère. Nous utilisons alors `Math.cos` et `Math.sin` pour déterminer l'influence que possède la poussée sur le mouvement horizontal et vertical du vaisseau :

```
    // Animer le vaisseau
    public function moveShip(timeDiff:uint) {

        // Pivoter et accélérer
        if (leftArrow) {
            ship.rotation -= shipRotationSpeed*timeDiff;
        } else if (rightArrow) {
            ship.rotation += shipRotationSpeed*timeDiff;
        } else if (upArrow) {
            shipMoveX += Math.cos(Math.PI*ship.rotation/180)*thrustPower;
            shipMoveY += Math.sin(Math.PI*ship.rotation/180)*thrustPower;
        }
```

Ensuite, la position du vaisseau est mise à jour en fonction de la vélocité :

```
        // Déplacer
        ship.x += shipMoveX;
        ship.y += shipMoveY;
```

L'une des particularités qui font tout l'attrait de ce jeu tient à la manière dont le vaisseau peut quitter l'écran d'un côté et réapparaître de l'autre. Voici le code qui se charge de gérer ce mécanisme. Un grand nombre de paramètres codés en dur pourraient ici être séparés et définis sous forme de constantes au début du script, mais le code est ainsi plus simple à lire et à comprendre.

L'écran fait 550 pixels de large et 400 de haut. Nous souhaitons que le vaisseau boucle non pas dès l'instant où il atteint le bord de l'écran mais juste au moment où on le perd de vue. À 570, le vaisseau revient ainsi à 590, ce qui le place à –20. Comme il se déplacerait vers la droite pour cela, il ne restera pas hors de vue.

Les 20 pixels supplémentaires que nous ajoutons aux bords de l'écran sont une zone morte du jeu. Vous ne pouvez pas voir les objets qui y transitent, et les missiles ne s'y trouvent pas non plus parce qu'ils disparaissent dès qu'ils atteignent le bord de l'écran.

Vous devez vous assurer que cette zone ne soit pas trop grande. Sans cela, de petits astéroïdes se déplaçant verticalement ou horizontalement pourraient disparaître pendant un certain temps. Vous pourriez aussi aisément y perdre votre vaisseau.

À l'inverse, si vous la rendez trop étroite, les objets sembleront se désintégrer sur un bord de l'écran et réapparaître miraculeusement sur l'autre.

```
    // Faire le tour de l'écran
    if ((shipMoveX > 0) && (ship.x > 570)) {
        ship.x -= 590;
    }
    if ((shipMoveX < 0) && (ship.x < -20)) {
        ship.x += 590;
    }
    if ((shipMoveY > 0) && (ship.y > 420)) {
        ship.y -= 440;
    }
    if ((shipMoveY < 0) && (ship.y < -20)) {
        ship.y += 440;
    }
}
```

Gérer les collisions du vaisseau

Lorsque le vaisseau est atteint par un astéroïde, il doit exploser. Pour cela, il passe à la troisième image intitulée `explode`. La fonction `removeAllShieldIcons` supprime les icônes de bouclier de l'écran. Un minuteur est ensuite configuré pour appeler `newShip` au bout de 2 secondes.

Le nombre de vaisseaux est réduit d'une unité et `removeShipIcon` est appelée afin de supprimer une icône de l'écran :

```
// Supprimer le vaisseau
public function shipHit() {
    gameMode = "delay";
    ship.gotoAndPlay("explode");
    removeAllShieldIcons();
    delayTimer = new Timer(2000,1);
    delayTimer.addEventListener(TimerEvent.TIMER_COMPLETE,newShip);
    delayTimer.start();
    removeShipIcon();
    shipsLeft--;
}
```

Levez vos boucliers !

Le bouclier est une partie un peu à part du vaisseau. Il figure sous la forme d'un clip à l'intérieur du clip `Ship`. Pour l'activer, il nous suffit donc de positionner sa propriété `visible` à `true`. Un minuteur est défini pour le désactiver au bout de 3 secondes. Entre-temps, `shieldOn` est positionné à `true` afin que toutes les collisions avec les astéroïdes soient ignorées.

Le bouclier est en fait un graphisme semi-transparent qui laisse transparaître le vaisseau. Un paramètre `alpha` est appliqué aux couleurs utilisées dans le dégradé du bouclier. Aucun code ActionScript n'est requis pour cela : le graphisme est simplement dessiné de cette manière.

La fonction `startShield` opère également une vérification au début et à la fin de la fonction. Au début, elle s'assure qu'il reste des boucliers au joueur. Ensuite, elle s'assure que le bouclier n'est pas déjà activé.

À la fin, elle vérifie le paramètre `freeShield`. S'il vaut `false`, nous réduisons le nombre de boucliers disponibles d'une unité et mettons à jour l'écran :

```
// Activer le bouclier pour 3 secondes
public function startShield(freeShield:Boolean) {
    if (shieldsLeft < 1) return; // Aucun bouclier restant
    if (shieldOn) return; // Bouclier déjà activé

    // Activer le bouclier et configurer le minuteur pour sa désactivation
    ship.shield.visible = true;
```

```
            shieldTimer = new Timer(3000,1);
            shieldTimer.addEventListener(TimerEvent.TIMER_COMPLETE,endShield);
            shieldTimer.start();

            // Mettre à jour le nombre de boucliers restants
            if (!freeShield) {
                removeShieldIcon();
                shieldsLeft--;
            }
            shieldOn = true;
        }
```

Lorsque le minuteur s'arrête, le bouclier est de nouveau rendu invisible et la variable booléenne `shieldOne` est positionnée à `false` :

```
        // Désactiver le bouclier
        public function endShield(event:TimerEvent) {
            ship.shield.visible = false;
            shieldOn = false;
        }
```

Astéroïdes

Viennent ensuite les fonctions qui gèrent les astéroïdes. Nous avons des fonctions pour créer des astéroïdes, pour les supprimer et pour les détruire.

Créer de nouveaux astéroïdes

Il existe trois tailles d'astéroïdes. Quand `newRock` est appelée, elle l'est avec le paramètre `rockType` qui spécifie la taille du nouvel astéroïde. Au début du jeu, tous les astéroïdes sont créés avec l'option de taille `"Big"`. Dans la suite du jeu, nous créons en revanche des paires d'astéroïdes à chaque frappe de missile qui utilisent les tailles `"Medium"` et `"Small"`.

Il existe pour chaque taille un `rockRadius` correspondant de `35`, `20` et `10`. Nous utiliserons ces nombres pour détecter les collisions par la suite.

Pour finir de créer l'astéroïde, une vélocité verticale est choisie en obtenant des valeurs aléatoires pour `dx` et `dy`. Nous récupérons aussi une valeur aléatoire pour `dr`, la vitesse de rotation.

L'un des autres éléments aléatoires est la variation des astéroïdes. Chaque clip contient trois images représentant chacune un astéroïde d'aspect différent.

Il serait intéressant de calculer dynamiquement le rayon de chaque astéroïde en testant les clips des astéroïdes eux-mêmes mais, à ce stade, nous n'en avons encore créé aucun et ne pouvons donc pas obtenir ces valeurs. Du reste, ce ne sont pas ces valeurs que nous souhaitons. Elles incluraient en effet les points les plus éloignés dans les graphismes, or nous souhaitons un nombre plus réduit qui donne une meilleure approximation du rayon global des astéroïdes.

Le tableau `rocks` est constitué d'objets de données incluant une référence au clip `rock`, les valeurs `dx`, `dy` et `dr`, ainsi que `rockType` (la taille) et `rockRadius` :

```
// Créer un unique astéroïde d'une taille spécifique
public function newRock(x,y:int, rockType:String) {

    // Créer la nouvelle classe appropriée
    var newRock:MovieClip;
    var rockRadius:Number;
    if (rockType == "Big") {
        newRock = new Rock_Big();
        rockRadius = 35;
    } else if (rockType == "Medium") {
        newRock = new Rock_Medium();
        rockRadius = 20;
    } else if (rockType == "Small") {
        newRock = new Rock_Small();
        rockRadius = 10;
    }

    // Choisir une apparence aléatoire
    newRock.gotoAndStop(Math.ceil(Math.random()*3));

    // Définir position de départ
    newRock.x = x;
    newRock.y = y;

    // Définir mouvement et rotation aléatoires
    var dx:Number = Math.random()*2.0-1.0;
    var dy:Number = Math.random()*2.0-1.0;
    var dr:Number = Math.random();

    // Ajouter à la scène et à la liste rocks
    gameObjects.addChild(newRock);
    rocks.push({rock:newRock, dx:dx, dy:dy, dr:dr, rockType:rockType, rockRadius: rockRadius});
}
```

Créer des vagues d'astéroïdes

Au début du jeu et à chaque nouvelle vague d'astéroïdes, nous appelons la fonction suivante pour créer quatre gros astéroïdes, tous espacés de manière égale à l'écran. La Figure 7.12 montre les positions des astéroïdes au tout début du jeu.

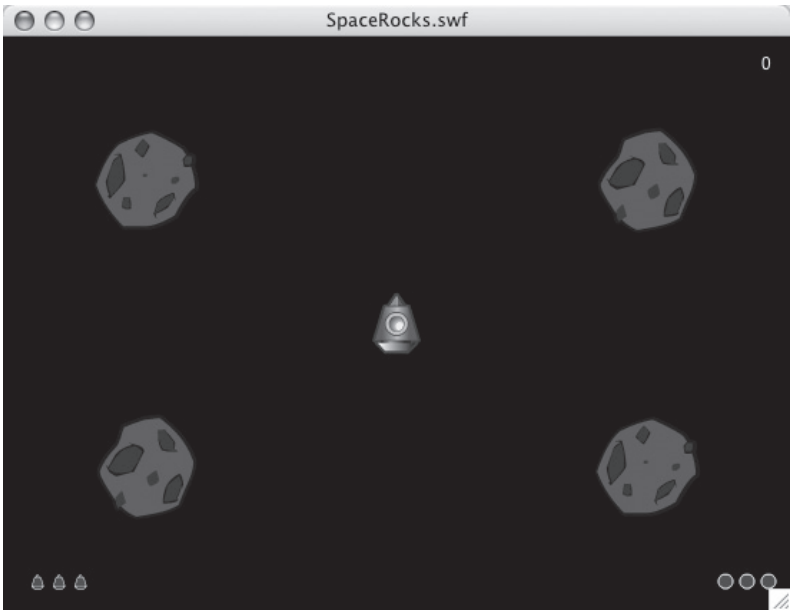


Figure 7.12
Les quatre astéroïdes sont placés à 100 pixels de distance des côtés et des bords supérieur et inférieur de l'écran.

Nous souhaitons donner à `gameMode` la valeur `play`. S'il s'agit de la première vague d'astéroïdes, nous avons déjà donné à `gameMode` la valeur `play`. S'il ne s'agit pas de la première vague, le `gameMode` doit avoir été modifié avec la valeur `delay` dans la fonction `shipHit`. Nous lui attribuons donc ici la valeur `play` par sécurité :

```
// Créer quatre astéroïdes
public function nextRockWave(event:TimerEvent) {
    rocks = new Array();
    newRock(100,100,"Big");
    newRock(100,300,"Big");
    newRock(450,100,"Big");
    newRock(450,300,"Big");
    gameMode = "play";
}
```

La fonction `newRockWave` crée à chaque fois quatre astéroïdes au même emplacement. Vous pourriez cependant souhaiter compliquer cette fonction en vérifiant la valeur de `gameLevel` et en utilisant des vagues de six astéroïdes au lieu de quatre pour les niveaux supérieurs à trois ou quatre. Voilà un bon moyen de pimenter le jeu. Il serait possible aussi d'ajouter des astéroïdes de petite et moyenne tailles dès le début de certains niveaux.

Déplacer les astéroïdes

Pour déplacer les astéroïdes, nous devons simplement examiner chacun des astéroïdes et obtenir les valeurs dans chaque objet du tableau `rocks`. La position est modifiée en fonction des valeurs `dx` et `dy`. La rotation est modifiée en fonction de la valeur `dr`.

Comme avec le vaisseau, nous devons faire passer les astéroïdes d'un côté de l'écran au côté opposé. Le code qui gère ce mouvement est pratiquement le même, les astéroïdes étant autorisés à sortir de 20 pixels en dehors de l'écran avant de faire leur boucle et de réapparaître côté opposé en ajoutant 40 pixels (20 de chaque côté) à la largeur de l'écran :

```
// Animer tous les astéroïdes
public function moveRocks(timeDiff:uint) {
    for(var i:int=rocks.length-1;i>=0;i--) {

        // Déplacer les astéroïdes
        var rockSpeed:Number = rockSpeedStart + rockSpeedIncrease*gameLevel;
        rocks[i].rock.x += rocks[i].dx*timeDiff*rockSpeed;
        rocks[i].rock.y += rocks[i].dy*timeDiff*rockSpeed;

        // Faire pivoter les astéroïdes
        rocks[i].rock.rotation += rocks[i].dr*timeDiff*rockSpeed;

        // Faire boucler les astéroïdes
        if ((rocks[i].dx > 0) && (rocks[i].rock.x > 570)) {
            rocks[i].rock.x -= 590;
        }
        if ((rocks[i].dx < 0) && (rocks[i].rock.x < -20)) {
            rocks[i].rock.x += 590;
        }
        if ((rocks[i].dy > 0) && (rocks[i].rock.y > 420)) {
            rocks[i].rock.y -= 440;
```

```
            }
            if ((rocks[i].dy < 0) && (rocks[i].rock.y < -20)) {
                rocks[i].rock.y += 440;
            }
        }
    }
```

Collisions des astéroïdes

Lorsqu'un astéroïde est touché, la fonction `rockHit` décide ce qu'elle doit en faire. Dans le cas d'un gros astéroïde, deux astéroïdes moyens sont créés à la place. Dans le cas d'un astéroïde moyen, deux petits astéroïdes le remplacent. Ils commencent au même emplacement que l'ancien, mais obtiennent des vitesses de rotation et des directions aléatoires.

Dans ces deux cas et dans le cas où c'est un petit astéroïde qui est touché, l'astéroïde d'origine est supprimé :

```
    public function rockHit(rockNum:uint) {
        // Créer deux astéroïdes plus petits
        if (rocks[rockNum].rockType == "Big") {
            newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Medium");
            newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Medium");
        } else if (rocks[rockNum].rockType == "Medium") {
            newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Small");
            newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Small");
        }
        // Supprimer l'astéroïde d'origine
        gameObjects.removeChild(rocks[rockNum].rock);
        rocks.splice(rockNum,1);
    }
```

Missiles

Les missiles sont créés lorsque le joueur appuie sur la barre d'espace. La fonction `newMissile` utilise la position du vaisseau pour lancer le missile et récupère en outre la rotation du vaisseau afin de déterminer la direction du missile.

Le missile n'est cependant pas positionné au centre du vaisseau ; il se trouve à un `shipRadius` de distance de ce centre, dans la même direction que celle que suivra le missile en s'éloignant. Ce réglage évite que les missiles paraissent provenir du centre du vaisseau.

Nous utilisons ici, pour simplifier les missiles, une astuce qui consiste à les représenter par un rond. Nous n'avons ainsi pas besoin de faire pivoter le missile selon un angle particulier. Les objets ronds possèdent la bonne apparence quelle que soit la direction dans laquelle ils se déplacent.

Nous tenons le registre de tous les missiles avec le tableau `missiles` :

```
// Créer un nouveau missile
public function newMissile() {
    // Créer
    var newMissile:Missile = new Missile();

    // Définir direction
    newMissile.dx = Math.cos(Math.PI*ship.rotation/180);
    newMissile.dy = Math.sin(Math.PI*ship.rotation/180);

    // Placement
    newMissile.x = ship.x + newMissile.dx*shipRadius;
    newMissile.y = ship.y + newMissile.dy*shipRadius;

    // Ajouter à la scène et au tableau
    gameObjects.addChild(newMissile);
    missiles.push(newMissile);
}
```

Lorsque les missiles se déplacent, nous utilisons la constante `missileSpeed` et `timeDiff` pour déterminer le nouvel emplacement.

Les missiles ne reviennent pas de l'autre côté de l'écran lorsqu'ils en sortent comme les astéroïdes et le vaisseau. Ils disparaissent simplement en quittant l'écran :

```
// Animer les missiles
public function moveMissiles(timeDiff:uint) {
    for(var i:int=missiles.length-1;i>=0;i--) {
        // Déplacer
        missiles[i].x += missiles[i].dx*missileSpeed*timeDiff;
        missiles[i].y += missiles[i].dy*missileSpeed*timeDiff;
        // Le bord de l'écran est passé
```

```
            if ((missiles[i].x < 0) || (missiles[i].x > 550) ||
                                        (missiles[i].y < 0) || (missiles[i].y > 400)) {
                gameObjects.removeChild(missiles[i]);
                missiles.splice(i,1);
            }
        }
    }
```

Lorsqu'un missile touche un astéroïde, il est également supprimé avec un appel à `missileHit` :

```
// Supprimer le missile
public function missileHit(missileNum:uint) {
    gameObjects.removeChild(missiles[missileNum]);
    missiles.splice(missileNum,1);
}
```

Nous supprimons les missiles dans `moveMissiles` à l'aide d'un bloc de code séparé au lieu d'appeler `missileHit`, par précaution pour l'avenir. Les deux événements se produisent dans des circonstances différentes. Si nous souhaitons que quelque chose de particulier se produise lorsque le missile atteint une cible, nous pourrons le placer dans `missileHit`. Or nous ne souhaiterions pas dans ce cas que le même événement se produise lorsque le missile a juste quitté l'écran.

Contrôle du jeu

Jusque-là, nous avons eu trois fonctions d'animation : `moveShip`, `moveRocks` et `moveMissiles`. Toutes trois sont appelées par la fonction d'animation principale, `moveGameObjects`. À son tour, cette dernière est appelée par l'événement `ENTER_FRAME` que nous avons configuré précédemment.

Déplacer les objets du jeu

La fonction `moveGameObjects` calcule le temps écoulé (`timePassed`) comme le faisait Air Raid et transmet cette valeur à ces trois fonctions. Notez que `moveShip` n'est appelée que si `gameMode` ne possède pas la valeur `"delay"`.

Pour finir, `moveGameObjects` appelle `checkCollisions`, qui est un organe central de notre jeu :

```
public function moveGameObjects(event:Event) {
    // Calculer le temps écoulé et animer
    var timePassed:uint = getTimer() - lastTime;
```

```
        lastTime += timePassed;
        moveRocks(timePassed);
        if (gameMode != "delay") {
            moveShip(timePassed);
        }
        moveMissiles(timePassed);
        checkCollisions();
    }
```

Vérifier les collisions

La fonction `checkCollisions` réalise les calculs critiques. Elle parcourt en boucle tous les astéroïdes et tous les missiles et vérifie si certains sont entrés en collision les uns avec les autres. Le `rockRadius` des astéroïdes est utilisé pour déterminer les collisions. Ce mécanisme est plus rapide que l'approche qui consiste à appeler `hitTestPoint`.

S'il y a une collision, les fonctions `rockHit` et `missileHit` sont appelées pour se charger des deux extrémités de la collision.

Si un astéroïde et un missile doivent être supprimés à ce stade, il est important qu'aucun d'entre eux ne soit plus testé pour les collisions possibles avec d'autres objets. Chacune des deux boucles `for` imbriquées se voit donc attribuer une étiquette. L'étiquette offre un moyen de spécifier à laquelle des boucles `for` une commande `break` ou `continue` est destinée. Dans le cas présent, nous souhaitons continuer dans la boucle `rockloop`, qui correspond à la boucle externe des boucles imbriquées. Une simple commande `break` amènerait le code à continuer à tester l'astéroïde pour voir s'il est entré en collision avec le vaisseau. Comme l'astéroïde n'existe plus, cette opération produirait une erreur :

```
    // Rechercher les missiles entrés en collision avec des astéroïdes
    public function checkCollisions() {
        // Parcourir les astéroïdes en boucle
        rockloop: for(var j:int=rocks.length-1;j>=0;j--) {
            // Parcourir les missiles en boucle
            missileloop: for(var i:int=missiles.length-1;i>=0;i--) {
                // Détection de collision
                if (Point.distance(new Point(rocks[j].rock.x,rocks[j].rock.y),
                        new Point(missiles[i].x,missiles[i].y))
                            < rocks[j].rockRadius) {

                    // Suppression de l'astéroïde et du missile
                    rockHit(j);
                    missileHit(i);
```

```
                // Mise à jour du score
                gameScore += 10;
                updateScore();

                // Nous sortons de cette boucle et reprenons la suivante
                continue rockloop;
            }
        }
```

Chaque astéroïde est testé afin de voir s'il entre en collision avec le vaisseau. Pour commencer, nous devons nous assurer que nous ne nous trouvons pas dans l'intervalle entre la disparition du vaisseau et sa réapparition. Nous devons aussi nous assurer que le bouclier est désactivé.

Si le vaisseau est touché, nous appelons les deux fonctions `shipHit` et `rockHit` :

```
        // Vérifier si un astéroïde heurte le vaisseau
        if (gameMode == "play") {
            if (shieldOn == false) { // only if shield is off
                if (Point.distance(new Point(rocks[j].rock.x,rocks[j].rock.y),
                        new Point(ship.x,ship.y))
                            < rocks[j].rockRadius+shipRadius) {

                    // Supprimer le vaisseau et l'astéroïde
                    shipHit();
                    rockHit(j);
                }
            }
        }
    }
```

Une fois que `checkCollisions` a terminé, elle examine rapidement le nombre d'astéroïdes à l'écran. Si tous les astéroïdes ont été supprimés, un minuteur est configuré pour lancer une nouvelle vague au bout de 2 secondes. `gameLevel` est incrémentée d'une unité, afin que les astéroïdes suivants soient plus rapides. `gameMode` se voit aussi attribuer la valeur `"betweenlevels"`. Cela signifie que la vérification ne sera pas réalisée avant que l'astéroïde ne réapparaisse, mais le joueur reste libre de déplacer le vaisseau :

```
    // Plus d'astéroïde, changer le mode du jeu et en recréer
    if ((rocks.length == 0) && (gameMode == "play")) {
        gameMode = "betweenlevels";
        gameLevel++; // Avancer d'un niveau
        delayTimer = new Timer(2000,1);
```

```
            delayTimer.addEventListener(TimerEvent.TIMER_COMPLETE,nextRockWave);
            delayTimer.start();
        }
    }
```

Terminer la partie

Si le vaisseau a été touché et qu'il n'en reste plus, la partie est terminée et la fonction `endGame` est appelée. Celle-ci s'occupe du nettoyage habituel et conduit l'animation à la troisième image du scénario :

```
    public function endGame() {
        // Supprimer tous les objets et écouteurs
        removeChild(gameObjects);
        removeChild(scoreObjects);
        gameObjects = null;
        scoreObjects = null;
        removeEventListener(Event.ENTER_FRAME,moveGameObjects);
        stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);

        gotoAndStop("gameover");
    }
```

Modifier le jeu

La fonctionnalité du bouclier dans ce jeu ne figure pas dans le jeu Asteroids original. On la trouve cependant dans ses variantes ultérieures et dans bien d'autres jeux du même genre. Le jeu d'origine contient pour sa part une autre fonctionnalité qui permet de faire disparaître le vaisseau pour le faire réapparaître à un emplacement aléatoire. Si cela conduit souvent le joueur à sa perte, c'est aussi un bon va-tout pour celui qui ne parvient pas à s'échapper d'une situation où il se trouve pris en étau.

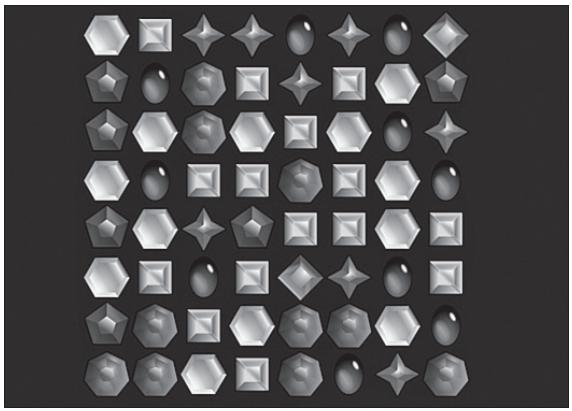
L'ajout de cette fonctionnalité est très simple : il suffit d'inclure un autre appui de touche dans `keyDownFunction` puis d'attribuer des valeurs `x` et `y` aléatoires au vaisseau.

Le jeu pourrait être amélioré avec des éléments simples comme des sons ou d'autres animations. La flamme du propulseur pourrait être animée en remplaçant le graphisme actuel par un symbole graphique en boucle dans le clip. Aucun code ActionScript n'est requis pour cela.

Vous pourriez également ajouter des vies supplémentaires : c'est une fonctionnalité bien courante dans ce type de jeu. Définissez simplement un palier pour le score, comme 1 000, et ajoutez une unité à `shipsLeft`. Vous devrez retracer les icônes des vaisseaux à ce moment, et pourquoi pas émettre un son pour signaler l'attribution du bonus.

La plupart des jeux de ce genre sur le Web ne sont pas des jeux dans l'espace. Ce concept général peut être utilisé à des fins éducatives ou commerciales en remplaçant les astéroïdes par des objets spécifiques. Par exemple, il peut s'agir de noms et de verbes, le joueur n'étant supposé tirer que sur les noms. Il pourrait aussi s'agir de bouts de papier à ramasser que vous êtes censé nettoyer.

Une modification simple pourrait consister à oublier complètement les missiles et à faire des collisions entre le vaisseau et les astéroïdes un objectif en soi. Vous pourriez ainsi récolter des objets au lieu de tirer dessus. Il pourrait notamment être intéressant de devoir récolter certains objets et en éviter d'autres.



8

Les “casual games” : Match Three

Au sommaire de ce chapitre :

- Classe réutilisable : PointBursts
- Match Three

À leurs débuts, les jeux vidéo étaient simples et amusants. L'un des plus populaires d'entre eux était ainsi un petit jeu de puzzle appelé Tetris. Les graphismes 3D ont ensuite repoussé les limites de cet univers et permis de créer des mondes virtuels en caméra subjective et de véritables jeux de rôle en ligne.

Les jeux de puzzle ont cependant connu un regain de succès au début de cette décennie avec les jeux en ligne gratuits et les jeux téléchargeables. On baptise en anglais cette catégorie de jeux les *casual games*.

L'appellation casual game n'est pas toujours très claire. Wikipédia en propose la définition suivante : "une catégorie de jeu électronique ou de jeu d'ordinateur ciblant le grand public". Cette définition est un peu large. On pourrait plus précisément parler de "jeux Match Three", car la plupart des sites Web qui proposent des casual games vendent principalement des jeux comme Match Three.

Bon nombre des jeux de ce livre correspondent cependant à la définition plus large. En fait, de nombreux puzzles d'images et de mots sont souvent vendus avec Match Three.

La plupart des *casual games* sont des jeux de puzzle d'action, autrement dit des jeux qui combinent un puzzle avec une sorte de mouvement ou un délai dans le temps pour pimenter le jeu.

Le jeu Match Three est de loin le plus courant des *casual games*. Près de la moitié des jeux sur les sites Web populaires de *casual games* sont des variantes de Match Three.

Dans ce chapitre, nous allons découvrir comment créer l'explosion de points, un effet spécial populaire utilisé dans les *casual games*. Ensuite, nous construirons un jeu Match Three classique.

Classe réutilisable : Point Bursts

Codes sources

http://flashgameu.com

A3GPU08_PointBurst.zip

Aux débuts des jeux d'arcades, lorsque le joueur réussissait une action, il se voyait attribuer des points. Cela n'a pas changé. Ce qui a changé, c'est le moyen de le signaler.

Dans les jeux d'arcade anciens, on ne voyait le score se modifier qu'au coin de l'écran. Malheureusement, le plus souvent, ce n'est pas à cet endroit que l'on regarde et l'on ne peut s'y reporter qu'une fois le feu de l'action bien passé. Il est donc judicieux que les jeux aient évolué et affichent le nombre de points gagnés directement à l'emplacement de l'écran où l'action se produit.

Vous découvrirez cela dans pratiquement tous les *casual games* pour le Web. La Figure 8.1 présente mon jeu Gold Strike juste au moment où le joueur clique sur des lingots d'or pour marquer des points. Vous pouvez voir le texte "30" à l'emplacement où se trouvaient auparavant les lingots d'or. Ces nombres grossissent en un instant puis disparaissent. Ils sont juste assez longtemps à l'écran pour faire remarquer au joueur le nombre de points qu'il a marqués.

Figure 8.1
Le nombre de points marqués s'affiche directement à l'endroit où se situe l'action.

J'appelle cet effet une *explosion de points*. Il est si courant et je l'utilise si fréquemment qu'il s'agit d'un excellent exemple de classe spéciale qui peut être créée puis réutilisée dans de nombreux jeux.

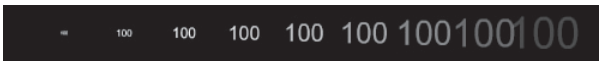
Développer la classe PointBurst

La classe **PointBurst.as** doit être aussi autonome que possible. En fait, notre but est de pouvoir utiliser une explosion de points avec une seule ligne de code dans le jeu. La classe elle-même a donc besoin de s'occuper de créer le texte et le sprite, de l'animer et de supprimer le tout par elle-même une fois qu'elle a terminé.

Notre classe PointBurst pourra être utilisée avec une seule ligne, mais ne nécessitera aucun élément dans la bibliothèque de l'animation principale à l'exception d'une police utilisée pour l'affichage des points.

La Figure 8.2 présente une version étalée dans le temps de ce que nous allons créer. L'explosion de points doit commencer par un score à taille réduite qui grossit progressivement. En outre, l'opacité doit être au départ de 100 % avant de s'effacer progressivement. Tout cela doit se passer en moins de 1 seconde.

Figure 8.2
Cette représentation synchronique montre à gauche le début de l'explosion de points puis chacun des stades de l'animation de gauche à droite.



Définition de la classe

Pour une classe si petite, il nous faut tout de même quatre instructions `import`. Nous utiliserons le minuteur pour contrôler l'animation de l'explosion de points, bien qu'une autre option eût été d'en faire une animation temporelle en utilisant des événements `ENTER_FRAME` :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
    import flash.utils.Timer;
```

Bien que la classe `PointBurst` réalise l'animation, il s'agit toujours d'un sprite, parce qu'elle ne requiert pas plusieurs images. Au lieu de cela, nous allons mettre à l'échelle et définir la propriété alpha du sprite à chaque étape.

Nous utiliserons des constantes statistiques pour déterminer le type de police, sa taille et sa couleur :

```
    public class PointBurst extends sprite {
        // Style de texte
        static const fontFace:String = "Arial";
        static const fontSize:int = 20;
        static const fontBold:Boolean = true;
        static const fontColor:Number = 0xFFFFFF;
```

Nous avons aussi plusieurs constantes associées à l'animation. `animSteps` et `animStepTime` déterminent la durée et la fluidité de l'animation. Par exemple, avec dix étapes et 50 millisecondes entre les étapes, nous obtenons une animation de 500 millisecondes. C'est encore le cas avec vingt étapes de 25 millisecondes, mais nous avons dans ce cas deux fois plus d'étapes et donc une animation plus fluide :

```
        // Animation
        static const animSteps:int = 10;
        static const animStepTime:int = 50;
```

L'échelle de l'animation change durant l'animation. Les deux constantes suivantes définissent les points de départ et de fin du changement d'échelle :

```
        static const startScale:Number = 0;
        static const endScale:Number = 2.0;
```

Après les constantes viennent plusieurs variables qui contiennent des références aux éléments dans l'explosion de points. L'une contient le champ texte et une autre, le `Sprite` qui encapsulera le champ texte. Une troisième contient une référence à la scène ou au clip où nous souhaitons placer l'explosion de points. La dernière variable contient une référence à l'objet `Timer` :

```
private var tField:TextField;
private var burstSprite:Sprite;
private var parentMC:MovieClip;
private var animTimer:Timer;
```

La fonction PointBurst

La ligne de code que nous utilisons pour créer un `PointBurst` (explosion de points) sert à créer un nouvel objet `PointBurst`. Cette opération appelle à son tour la fonction `PointBurst`, qui accepte des paramètres. Ces paramètres sont notre seul moyen de communiquer à l'objet `PointBurst` des informations clés telles que l'emplacement de l'explosion de points et le texte à afficher.

Le paramètre `pts` est un `Object` car nous souhaitons pouvoir accepter n'importe quel type de variable : `int`, `Number` ou `String`. Nous convertirons cela par la suite en `String`, car c'est ce que requiert la propriété `text` du `TextField`.

Le premier paramètre de `PointBurst` est un clip, `mc`. Il s'agira d'une référence à la scène ou à un autre clip ou un sprite auquel l'explosion de points sera ajoutée avec `addChild` :

```
public function PointBurst(mc:MovieClip, pts:Object, x,y:Number) {
```

La première chose que la fonction doit faire est de créer un objet `TextFormat` à attribuer au `TextField` que nous allons créer par la suite. Nous utiliserons pour cela des constantes de mise en forme que nous aurons définies antérieurement. La fonction devra aussi définir l'alignement du champ en choisissant `center` :

```
// Création du format de texte
var tFormat:TextFormat = new TextFormat();
tFormat.font = fontFace;
tFormat.size = fontSize;
tFormat.bold = fontBold;
tFormat.color = fontColor;
tFormat.align = "center";
```

Ensuite, nous créons le `TextField` lui-même. En plus de positionner `selectable` à `false`, nous devons demander au champ d'utiliser des polices incorporées au lieu de polices système. Nous souhaitons en effet définir la transparence du texte, ce qui ne peut être réalisé que si le texte utilise des polices incorporées.

Pour centrer le texte dans le sprite que nous allons créer ensuite, nous positionnons la propriété `autoSize` du champ à `TextFieldAutoSize.CENTER`. Ensuite, nous positionnons les propriétés `x` et `y` à l'inverse de la moitié de la largeur et de la hauteur. Cela place le centre du texte au point 0,0 :

```
// Créer le champ texte
tField = new TextField();
tField.embedFonts = true;
tField.selectable = false;
tField.defaultTextFormat = tFormat;
tField.autoSize = TextFieldAutoSize.CENTER;
tField.text = String(pts);
tField.x = -(tField.width/2);
tField.y = -(tField.height/2);
```

Nous créons maintenant un sprite pour contenir le texte et agir comme objet d'affichage principal pour l'animation. Nous définissons l'emplacement de ce sprite en lui attribuant les valeurs `x` et `y` passées dans la fonction. Nous définissons l'échelle du sprite en lui attribuant la constante `startScale`. Nous fixons la propriété `alpha` à zéro. Ensuite, nous ajoutons le sprite au clip `mc`, c'est-à-dire au sprite passé dans la fonction :

```
// Création du sprite
burstSprite = new Sprite();
burstSprite.x = x;
burstSprite.y = y;
burstSprite.scaleX = startScale;
burstSprite.scaleY = startScale;
burstSprite.alpha = 0;
burstSprite.addChild(tField);
parentMC = mc;
parentMC.addChild(burstSprite);
```

Maintenant que l'objet `PointBurst` s'est manifesté comme sprite, nous devons simplement démarrer un minuteur pour lancer l'animation au cours des 500 prochaines millisecondes. Ce minuteur appelle `rescaleBurst` plusieurs fois, puis appelle `removeBurst` une fois que c'est fini :

```
        // Démarrer l'animation
        animTimer = new Timer(animStepTime,animSteps);
        animTimer.addEventListener(TimerEvent.TIMER, rescaleBurst);
        animTimer.addEventListener(TimerEvent.TIMER_COMPLETE, removeBurst);
        animTimer.start();
    }
```

Animer l'explosion de points

Lorsque le `Timer` appelle `rescaleBurst`, nous devons définir les propriétés d'échelle et l'alpha du sprite. Pour commencer, nous calculons `percentDone` en fonction du nombre d'étapes de `Timer` qui se sont écoulées et du nombre total d'étapes `animSteps`. Ensuite, nous appliquons cette valeur aux constantes `startScale` et `endScale` pour obtenir l'échelle actuelle. Nous pouvons utiliser `percentDone` pour définir la propriété `alpha`, mais nous souhaitons inverser la valeur afin d'aller de `1.0` à `0.0`.

La propriété `alpha` définit la transparence d'un sprite ou d'un clip. À `1.0`, l'objet se comporte normalement, les couleurs unies étant à 100 % d'opacité. Cela signifie cependant que les zones non remplies, comme celles en dehors de la forme des caractères, sont quant à elles transparentes. À `.5` ou 50 % de transparence, les zones habituellement opaques comme les contours et les remplissages des caractères partagent les pixels avec les couleurs sous-jacentes.

```
// Animer
public function rescaleBurst(event:TimerEvent) {
    // À quel point en sommes-nous ?
    var percentDone:Number = event.target.currentCount/animSteps;
    // Définir échelle et alpha
    burstSprite.scaleX = (1.0-percentDone)*startScale + percentDone*endScale;
    burstSprite.scaleY = (1.0-percentDone)*startScale + percentDone*endScale;
    burstSprite.alpha = 1.0-percentDone;
}
```

Lorsque le `Timer` a fini, il appelle `removeBurst`. Cette fonction s'occupe de tout pour que le `PointBurst` se débarrasse lui-même, sans aucune action de la part de l'animation principale ou de la classe de l'animation.

Une fois que le champ texte `tField` est supprimé du sprite `burstSprite`, `burstSprite` est supprimé de `parentMC`. Ensuite, les deux sont positionnés à `null` afin de les nettoyer de la mémoire. Pour finir, la commande `delete` est utilisée pour faire complètement disparaître l'objet `PointBurst`.

On ne peut pas dire avec certitude que toutes les lignes de `removeBurst` sont nécessaires. Normalement, vous êtes censé effacer toutes les références à un objet pour le supprimer, mais l'instruction `delete` supprime le `PointBurst`, qui à son tour doit supprimer les deux variables également. La suppression de `burstSprite` peut aussi permettre de supprimer `tField`. Il n'existe pas de moyen de le tester et, à l'heure où ces lignes sont écrites, il ne semble pas exister de document technique qui indique ce que le lecteur Flash fait spécifiquement dans ce cas. Il est donc préférable d'utiliser une fonction qui s'assure que tout est correctement supprimé.

```
// Terminé, se supprimer
public function removeBurst(event:TimerEvent) {
    burstSprite.removeChild(tField);
    parentMC.removeChild(burstSprite);
    tField = null;
    burstSprite = null;
    delete this;
}
```

Utiliser PointBurst dans une animation

Vous aurez besoin de deux choses avant de créer un nouvel objet `PointBurst` dans une animation. La première est de créer une nouvelle police dans la bibliothèque de l'animation. La seconde est d'indiquer à Flash où trouver votre fichier **PointBurst.as**.

Ajouter une police à une animation

Il faut une police parce que nous utilisons la propriété `alpha` pour la transparence du texte et cela ne peut se faire qu'avec une police incorporée dans la bibliothèque.

Pour créer une police incorporée, vous devez utiliser le menu déroulant du panneau Bibliothèque et choisir Nouvelle police. Ensuite, sélectionnez la police souhaitée et nommez-la. Le nom de l'élément dans la bibliothèque n'importe pas vraiment. J'utilise généralement le nom de la police, comme Arial dans cet exemple. La Figure 8.3 présente la boîte de dialogue Propriétés des symboles de police.

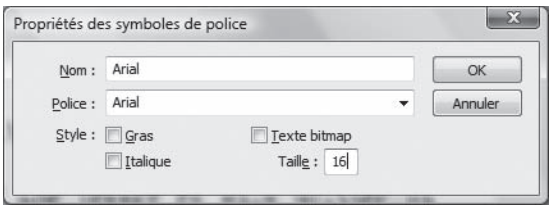


Figure 8.3
La boîte de dialogue Propriétés des symboles de police vous permet de choisir une police à ajouter à la bibliothèque.

Il ne s'agit là que d'une étape. L'étape numéro deux, qui n'a rien d'évident, consiste à s'assurer que cette police est incluse pour qu'ActionScript puisse l'utiliser. Pour cela, sélectionnez la police dans la bibliothèque, puis cliquez du bouton droit ou cliquez avec la touche Ctrl enfoncée sur Mac et sélectionnez l'option Liaison pour faire apparaître la boîte de dialogue Propriétés de liaison présentée à la Figure 8.4.

À la différence des autres types de symboles, les options de liaisons n'apparaissent pas dans la boîte de dialogue Propriétés générales du symbole. Elles ne s'affichent que dans la boîte de dialogue Propriétés de liaison.

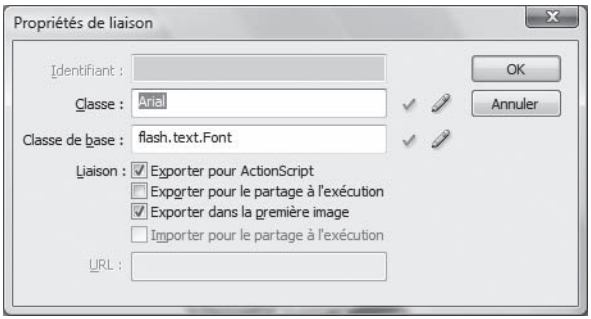


Figure 8.4
La boîte de dialogue Propriétés de liaison vous permet de spécifier une classe pour la police dans la bibliothèque.

Le nom de classe attribué à la police n'importe pas. Vous pouvez en outre ignorer la boîte de dialogue qui vous indique qu'aucune définition de classe n'est trouvée. Il n'en faut aucune.

Emplacements des classes

Pour nos exemples, nous n'aurons rien à faire pour indiquer à Flash où rechercher le fichier de classe **PointBurst.as**, car celui-ci se trouve au même emplacement que l'animation Flash.

En revanche, si vous souhaitez utiliser le même fichier de classe **PointBurst.as** dans plusieurs projets, vous devez le placer à un endroit auquel peuvent accéder toutes les animations et leur indiquer à chacune où le trouver.

Il existe deux moyens de procéder. Le premier consiste à ajouter un chemin de classe aux préférences de Flash. Il se peut que vous souhaitiez créer un dossier pour contenir toutes les classes que vous utilisez régulièrement. Ensuite, accédez à la section ActionScript des Préférences Flash. Cliquez sur le bouton Paramètres ActionScript 3.0 et ajoutez un dossier à l'emplacement que consulte Flash pour les fichiers de classe.

Vous pouvez aussi utiliser tout simplement l'emplacement par défaut pour les classes de bibliothèque, soit le dossier Classes, qui se trouve dans le dossier Flash du dossier Program Files ou Applications. Pour ma part, j'évite cette approche car je m'efforce de conserver tous les documents que je crée en dehors du dossier Applications, afin de n'y conserver que l'installation par défaut de mes applications.

L'autre méthode pour indiquer à une animation où trouver un fichier de classe qui ne se trouve pas dans le même répertoire que l'animation consiste à choisir Fichier > Paramètres de publication, puis à cliquer sur le bouton Paramètres à côté de la sélection de la version ActionScript. Vous pourrez alors ajouter un nouveau chemin de classe pour cette seule animation.

En résumé, il existe quatre moyens pour une animation Flash d'accéder à un fichier de classe :

1. Placer le fichier de classe dans le même dossier que l'animation.
2. Ajouter l'emplacement de classe dans les préférences de Flash.
3. Placer le fichier de classe dans le dossier Classes de l'application Flash.
4. Ajouter l'emplacement de la classe dans les Paramètres de publication de l'animation.

Créer une explosion de points

Une fois que la police se trouve dans la bibliothèque et que l'animation a accès à la classe, il suffit d'une ligne pour créer une explosion de points. Voici un exemple :

```
var pb:PointBurst = new PointBurst(this,100,50,75);
```

Ce code crée une explosion de points qui affiche le nombre 100. L'explosion se produit aux coordonnées `50,75`.

L'animation d'exemple **PointBurstExample.fla** et sa classe **PointBurstExample.as** correspondante présentent un exemple légèrement plus évolué. Il crée une explosion de points partout où vous cliquez :

```
package {
    import flash.display.*;
    import flash.events.*;

    public class PointBurstExample extends MovieClip {

        public function PointBurstExample() {
            stage.addEventListener(MouseEvent.CLICK,tryPointBurst);
        }

        public function tryPointBurst(event:MouseEvent) {
            var pb:PointBurst = new PointBurst(this,100,mouseX,mouseY);
        }
    }
}
```

Maintenant que nous disposons d'un fragment de code indépendant qui s'occupe de cet effet spécial assez complexe, nous pouvons passer à notre jeu suivant en sachant qu'il pourra inclure l'explosion de points en ne levant presque pas le petit doigt pour programmer.

Match Three

Codes sources
http://flashgameu.com
A3GPU08_MatchThree.zip

Si Match Three est le *casual game* le plus courant et le plus populaire, il ne l'est pas devenu parce qu'il est facile à programmer. En réalité, bien des aspects de ce jeu requièrent des techniques assez sophistiquées. Nous allons également examiner ce jeu élément par élément.

Jouer à Match Three

Si vous êtes parvenu à ne pas jouer à des jeux Match Three au cours des quelques dernières années, voici le principe du jeu.

Une grille de huit par huit contient une disposition aléatoire de six ou sept pièces de jeu. Vous pouvez cliquer sur n'importe quelle paire de pièces adjacentes pour tenter de les permuter. Si la permutation produit un alignement horizontal ou vertical de trois pièces identiques ou plus, elle est autorisée. Les pièces qui s'alignent sont supprimées et celles qui se trouvent au-dessus tombent dans l'emplacement libéré. De nouvelles pièces viennent du haut afin de remplir le trou laissé par la disparition des pièces alignées.

Voilà tout. C'est cette simplicité du jeu qui le rend si populaire. Le jeu se poursuit jusqu'à ce que la grille parvienne à un état où aucune permutation n'est plus possible.

La Figure 8.5 présente mon jeu Newton's Nightmare, un jeu Match Three plutôt classique.

Figure 8.5
Le jeu Newton's Nightmare utilise des pommes pour ce jeu de Match Three.

Le jeu Bejeweled, aussi appelé Diamond Mine, est réputé avoir lancé la fureur du Match Three.

Vue d'ensemble des fonctionnalités du jeu

La séquence d'événement dans le jeu suit douze étapes. Chaque étape présente un défi différent en termes de programmation.

1. Créer une grille aléatoire

Une grille de huit par huit avec une disposition aléatoire de sept éléments différents est créée au début du jeu.

2. Vérifier les correspondances

Il existe quelques restrictions concernant ce que la grille de départ peut contenir. La première tient à ce qu'elle ne peut inclure de lignes avec des alignements de trois éléments. Il faut que le joueur lui-même trouve la première correspondance permettant de créer un alignement.

3. Vérifier les déplacements

La seconde restriction pour la grille initiale tient à ce qu'il doit exister au moins un déplacement valide. Autrement dit, le joueur doit pouvoir permuter deux éléments et obtenir une correspondance.

4. Le joueur sélectionne deux pièces

Les pièces doivent être adjacentes l'une à l'autre horizontalement ou verticalement et la permutation doit produire une correspondance.

5. Les pièces sont permutées

En général, une animation montre les deux pièces qui se déplacent afin d'occuper leurs emplacements respectifs.

6. Rechercher des correspondances

Une fois qu'une permutation est effectuée, il faut examiner la grille à la recherche de nouvelles correspondances pour les alignements horizontaux et verticaux. Si aucune correspondance n'est trouvée, la permutation doit être annulée.

7. Attribuer des points

Si une correspondance est trouvée, des points doivent être attribués.

8. Supprimer les correspondances

Les pièces impliquées dans une correspondance doivent être supprimées de la grille.

9. Faire tomber les pièces du dessus

Les pièces situées au-dessus de celles qui ont été supprimées doivent tomber pour remplir l'espace laissé vide.

10. Ajouter de nouvelles pièces

De nouvelles pièces doivent tomber du haut de la grille afin de remplir les espaces vacants.

11. Rechercher des correspondances de nouveau

Une fois que toutes les pièces sont tombées et que de nouvelles sont venues remplir les trous, il convient de rechercher de nouvelles correspondances. Retour à l'étape 6.

12. Vérifier si d'autres déplacements sont possibles

Avant de redonner le contrôle au joueur, une vérification est opérée afin de voir si d'autres déplacements sont possibles. Si ce n'est pas le cas, la partie est terminée.

L'animation et la classe *MatchThree*

L'animation **MatchThree.fla** est assez simple. Hormis la police Arial dans la bibliothèque, les seuls éléments liés au jeu sont un clip pour les pièces du jeu et un autre clip qui agit comme indicateur de sélection.

La Figure 8.6 présente le clip `Piece`. Celui-ci contient sept images, chacune abritant une pièce différente. Le clip de sélection se trouve également sur le calque supérieur et s'étend sur les sept images. Il peut être activé ou désactivé à l'aide de la propriété `visible`.

Examinons les définitions de classe avant de passer à la logique du jeu. Bizarrement, il n'y a pas autant de définitions que l'on pourrait en attendre. Seules les instructions `import` les plus élémentaires sont requises :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
    import flash.utils.Timer;
```

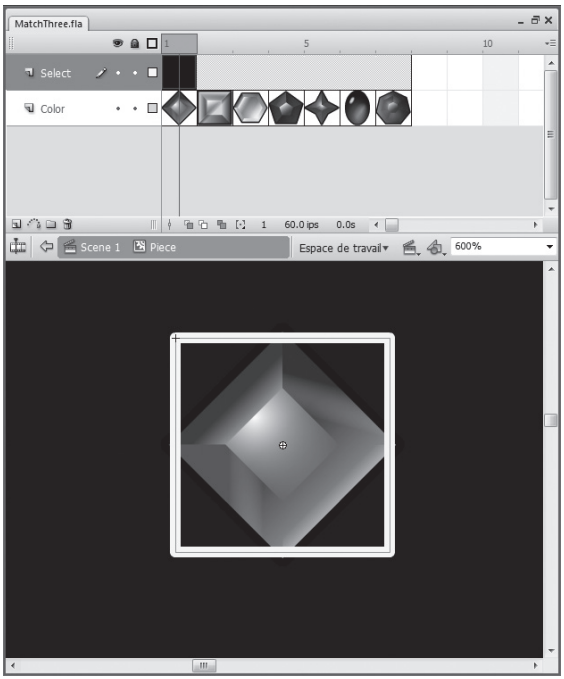


Figure 8.6
Le clip `Piece` contient sept variantes et un cadre de sélection.

En ce qui concerne les constantes, nous n'en utilisons qu'une pour le nombre de variantes de `Piece` et trois pour la disposition de l'affichage à l'écran :

```
public class MatchThree extends MovieClip {
    // Constantes
    static const numPieces:uint = 7;
    static const spacing:Number = 45;
    static const offsetX:Number = 120;
    static const offsetY:Number = 30;
```

L'état du jeu sera stocké dans cinq variables différentes. La première, `grid`, contient des références à tous les objets `Piece`. Il s'agit en fait d'un tableau de tableaux. Chaque élément dans la grille correspond ainsi en réalité à un autre tableau contenant huit références au clip `Piece`. Il s'agit donc d'un tableau imbriqué de huit par huit. Nous pouvons dès lors examiner chaque objet `Piece` en utilisant simplement `grid[x][y]`.

`gameSprite` est un sprite destiné à contenir tous les sprites et les clips que nous allons créer, ce qui permet de les séparer de tous les autres graphismes déjà sur la scène.

La variable `firstPiece` contient une référence au premier objet `Piece` sur lequel le joueur clique, comme pour le jeu de Memory du Chapitre 3.

Les deux variables booléennes `isDropping` et `isSwapping` indiquent si des objets `Piece` sont en cours d'animation sur le moment. La variable `gameScore` contient le score du joueur :

```
// Grille et mode du jeu
private var grid:Array;
private var gameSprite:Sprite;
private var firstPiece:Piece;
private var isDropping,isSwapping:Boolean;
private var gameScore:int;
```

Configurer la grille

Les premières fonctions définiront les variables du jeu, notamment en configurant la grille du jeu.

Définir les variables du jeu

Pour commencer la partie, nous devons définir toutes les variables d'état du jeu. Nous commencerons par créer le tableau de tableaux `grid`. Ensuite, nous appellerons `setUpGrid` pour le remplir.

Il n'est pas nécessaire de remplir les tableaux internes de `grid` par des emplacements vides. Le simple fait de définir une position dans un tableau crée l'emplacement correspondant et remplit tous les emplacements précédents avec la valeur `undefined`. Par exemple, si un nouveau tableau est créé puis que l'élément 3 se voie attribuer la valeur `"My String"`, le tableau possède la longueur 3 et les éléments 0 et 1 possèdent la valeur `undefined`.

Nous définissons ensuite les variables `isDropping`, `isSwapping` et `gameScore`. Nous configurons en outre un écouteur `ENTER_FRAME` pour diriger les mouvements dans le jeu :

```
// Configurer la grille et début de la partie
public function startMatchThree() {
    // Créer le tableau grid
    grid = new Array();
    for(var gridrows:int=0;gridrows<8;gridrows++) {
        grid.push(new Array());
    }
    setUpGrid();
    isDropping = false;
    isSwapping = false;
    gameScore = 0;
    addEventListener(Event.ENTER_FRAME,movePieces);
}
```

Configuration de la grille

Pour configurer la grille, nous commençons par une boucle sans fin en utilisant l'instruction `while(true)`. Ensuite, nous créons les éléments dans la grille. Nous prévoyons que la première tentative créera une grille valide.

Un nouveau `gameSprite` est créé pour contenir les clips pour les pièces du jeu. Ensuite, soixante-quatre objets `Piece` aléatoires sont créés avec la fonction `addPiece`. Nous examinerons cette fonction ensuite mais, pour l'instant, sachez qu'elle ajoute un objet `Piece` au tableau `grid` ainsi qu'au `gameSprite` :

```
public function setUpGrid() {
    // Boucler jusqu'à ce que la grille de départ soit valide
    while (true) {
        // Créer sprite
        gameSprite = new Sprite();

        // Ajouter 64 pièces aléatoires
        for(var col:int=0;col<8;col++) {
            for(var row:int=0;row<8;row++) {
                addPiece(col,row);
            }
        }
```

Ensuite, nous devons vérifier deux choses pour déterminer si la grille créée est un point de départ valide. La fonction `lookForMatches` retourne un tableau des correspondances trouvées. Nous l'examinerons plus loin dans ce chapitre. À ce stade, elle doit retourner zéro, ce qui signifie qu'il n'y a pas de correspondance à l'écran. Une commande `continue` ignore le reste de la boucle `while` et recommence en créant une nouvelle grille.

Après cela, nous appelons `lookForPossibles`, qui vérifie s'il existe des correspondances possibles suite à une permutation. Si la fonction retourne `false`, il ne s'agit pas d'un bon point de départ parce que le jeu est déjà terminé.

Si aucune de ces conditions n'est remplie, la commande `break` permet au programme de quitter la boucle `while`. Ensuite, nous ajoutons le `gameSprite` à la scène :

```
        // Essayer à nouveau si des correspondances sont présentes
        if (lookForMatches().length != 0) continue;

        // Essayer à nouveau si aucun déplacement possible
        if (lookForPossibles() == false) continue;

        // Aucune correspondance, mais des coups sont possibles : la grille est OK
        break;
    }

    // Ajouter le sprite
    addChild(gameSprite);
}
```

Ajouter les pièces du jeu

La fonction `addPiece` crée un objet `Piece` aléatoire à un emplacement de colonne et de ligne défini. Elle crée le clip et définit son emplacement :

```
// Créer une pièce aléatoire, ajouter au sprite et à la grille
public function addPiece(col,row:int):Piece {
    var newPiece:Piece = new Piece();
    newPiece.x = col*spacing+offsetX;
    newPiece.y = row*spacing+offsetY;
```

Chaque `Piece` doit tenir le registre de son propre emplacement sur la grille. Deux propriétés dynamiques appelées `col` et `row` sont définies à cet effet. En outre, `type` contient le nombre correspondant au type de l'objet `Piece`, qui correspond à l'image du clip qui est affichée :

```
    newPiece.col = col;
    newPiece.row = row;
    newPiece.type = Math.ceil(Math.random()*7);
    newPiece.gotoAndStop(newPiece.type);
```

Le clip `select` à l'intérieur du clip `Piece` correspond au contour qui apparaît lorsque l'utilisateur clique sur une pièce. Nous le définirons de manière qu'il ne soit pas visible au départ. Ensuite, la `Piece` est ajoutée au sprite `gameSprite`.

Pour placer la `Piece` dans le tableau `grid`, nous utilisons l'adressage des tableaux imbriqués à l'aide des doubles crochets : `grid[col][row] = newPiece`.

Chaque `Piece` se voit attribuer son propre écouteur de clic. Ensuite, la référence à l'objet `Piece` est retournée. Nous l'utiliserons non pas dans la fonction `setUpGrid` précédente mais plus tard lors de la création de nouveaux objets `Piece` servant à remplacer ceux qui ont été supprimés suite à une correspondance :

```
    newPiece.select.visible = false;
    gameSprite.addChild(newPiece);
    grid[col][row] = newPiece;
    newPiece.addEventListener(MouseEvent.CLICK,clickPiece);
    return newPiece;
}
```

La Figure 8.7 présente une grille valide aléatoire terminée.

Interaction du joueur

Lorsque le joueur clique sur un objet `Piece`, l'action résultante dépend du fait qu'il s'agit de la première pièce sur laquelle il clique ou de la seconde. S'il s'agit de la première, l'objet `Piece` est sélectionné et rien d'autre ne se passe.

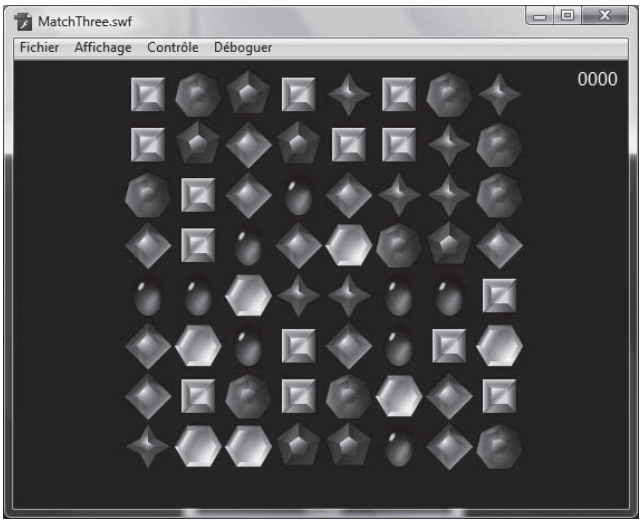


Figure 8.7
Voici l'une des grilles de jeu aléatoires possibles.

Si le joueur clique deux fois sur le même objet `Piece`, celui-ci est désélectionné et le joueur revient à la case départ :

```
// Le joueur clique sur une pièce
public function clickPiece(event:MouseEvent) {
    var piece:Piece = Piece(event.currentTarget);

    // La première est sélectionnée
    if (firstPiece == null) {
        piece.select.visible = true;
        firstPiece = piece;

    // Le joueur a cliqué de nouveau sur la première
    } else if (firstPiece == piece) {
        piece.select.visible = false;
        firstPiece = null;
```

Si le joueur a cliqué sur un deuxième objet `Piece`, nous devons déterminer s'il peut y avoir une permutation. Pour commencer, nous désactivons la mise en surbrillance de la sélection sur le premier objet `Piece`.

Le premier test consiste à déterminer si les deux objets `Piece` se trouvent sur la même ligne, puis s'ils se trouvent côte à côte. Sinon les objets `Piece` peuvent être dans la même colonne et l'un au-dessus de l'autre.

Dans les deux cas, nous appelons `makeSwap`. Cette fonction s'occupe de vérifier si la permutation est valide – autrement dit, si elle produira un alignement. Que ce soit le cas ou non, la variable `firstPiece` est positionnée à `null` afin de se préparer à la prochaine sélection du joueur.

Si le joueur a sélectionné un objet `Piece` qui se trouve trop éloigné du premier, nous supposons que le joueur souhaite abandonner sa première sélection et commençons à sélectionner une seconde paire :

```
            // Le joueur a cliqué sur la seconde pièce
            } else {
                firstPiece.select.visible = false;

                // Même ligne, une colonne après
                if (firstPiece.row == piece.row) {
                    if (Math.abs(firstPiece.col-piece.col) == 1) {
                        makeSwap(firstPiece,piece);
                        firstPiece = null;
                    }

                // Même colonne, une ligne après
                } else if (firstPiece.col == piece.col) {
                    if (Math.abs(firstPiece.row-piece.row) == 1) {
                        makeSwap(firstPiece,piece);
                        firstPiece = null;
                    }

                // Mauvais déplacement, resélection de la première pièce
                } else {
                    firstPiece = piece;
                    firstPiece.select.visible = true;
                }
            }
        }
```

La fonction `makeSwap` permute les deux objets `Piece` puis vérifie si une correspondance est disponible. Si ce n'est pas le cas, elle repermute les objets `Piece`. Sinon la variable `isSwapping` est positionnée à `true` afin que l'animation puisse s'exécuter :

```
        // Lancer la permutation animée des deux pièces
        public function makeSwap(piece1,piece2:Piece) {
            swapPieces(piece1,piece2);

            // Vérifier si le déplacement a produit un alignement
            if (lookForMatches().length == 0) {
                swapPieces(piece1,piece2);
            } else {
                isSwapping = true;
            }
        }
```

Pour effectuer la permutation effective, nous devons stocker l'emplacement de la première `Piece` dans une variable temporaire. Ensuite, nous définirons l'emplacement de la première `Piece` en lui attribuant celui de la seconde. La Figure 8.8 présente un diagramme du fonctionnement de la permutation.

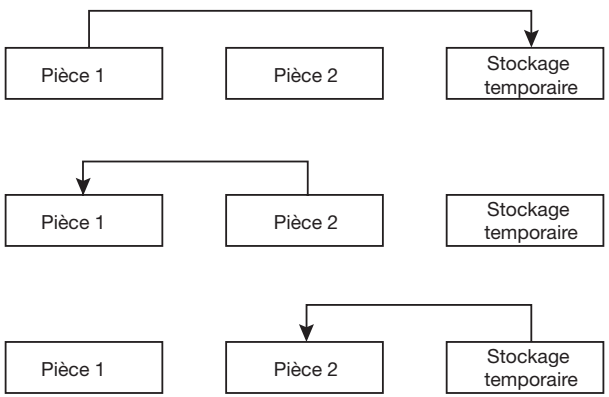


Figure 8.8
Lors de la permutation de deux valeurs, il est nécessaire de créer une variable temporaire pour stocker une valeur au cours de l'échange.

Une fois les emplacements des objets `Piece` échangés, la grille doit être mise à jour. Comme chaque `Piece` possède maintenant les valeurs `row` et `col` appropriées, nous configurons simplement la grille de manière à ponter sur chaque `Piece` au bon emplacement dans la grille :

```
// Permuter les deux pièces
public function swapPieces(piece1,piece2:Piece) {
    // Permuter les valeurs row et col
    var tempCol:uint = piece1.col;
    var tempRow:uint = piece1.row;
    piece1.col = piece2.col;
    piece1.row = piece2.row;
    piece2.col = tempCol;
    piece2.row = tempRow;

    // Permuter les positions dans grid
    grid[piece1.col][piece1.row] = piece1;
    grid[piece2.col][piece2.row] = piece2;
}
```

La permutation est complètement réversible, ce qui est important car elle devra souvent être inversée. Nous ne savons si la permutation produit un alignement concordant qu'après qu'elle se trouve réalisée. Nous aurons ainsi souvent besoin de permuter les objets `Piece`, de vérifier s'il existe un alignement, puis de reprendre la permutation si aucun alignement n'est trouvé.

Animer le mouvement des pièces

Nous allons utiliser une méthode intéressante mais pas évidente pour l'animation du mouvement des pièces. Chaque objet `Piece` connaît la ligne (`row`) et la colonne (`col`) dans laquelle il se trouve parce qu'il possède une propriété `row` et une propriété `col` dynamiques. Il connaît également son emplacement à l'écran grâce à ses propriétés `x` et `y`.

Ces deux positions doivent se correspondre avec l'aide des variables `spacing`, `offsetX` et `offsetY`. Une `Piece` dans la colonne 3 doit ainsi se trouver à la position `x = 3*spacing+offsetX`.

Que se passe-t-il cependant si une `Piece` se déplace vers une nouvelle colonne ? Si nous fixons la valeur de `col` d'une `Piece` à `4`, elle devrait être à `4*spacing+offsetX`, qui est située `spacing` (45) pixels vers la droite. Dans le cas présent, nous demandons à la `Piece` de se déplacer un peu vers la droite, afin de se rapprocher de son nouvel emplacement. Si nous procédons ainsi à chaque image, la `Piece` parviendra à terme à sa nouvelle destination et cessera de bouger (car elle possédera une valeur `col` et une valeur `x` qui se correspondent).

Cette technique nous permet d'animer n'importe quel objet `Piece` pendant qu'il se déplace vers son nouvel emplacement. Nous n'avons même pas besoin de configurer cette animation pièce par pièce : il nous suffit de changer la propriété `col` ou `row` d'une `Piece` pour que la fonction suivante s'occupe du reste.

La fonction `movePieces` est appelée à chaque événement `ENTER_FRAME` car nous l'avons configurée avec un écouteur au début de la classe. Elle parcourt en boucle toutes les pièces et vérifie toutes les valeurs `col` et `row` afin de voir si les valeurs `x` et `y` doivent être ajustées pour y correspondre.

Nous utilisons une distance de `5` à chaque image dans `movePieces`. Pour que `col` et `row` s'alignent sur les valeurs `x` et `y`, il est impératif de s'en tenir à des multiples de 5 pour `spacing`. Dans l'animation d'exemple, `spacing` vaut `45` et cela fonctionne. Si vous deviez changer la valeur de `spacing`, par exemple en choisissant `48`, vous devriez également choisir une nouvelle quantité de mouvement dont 48 soit un multiple, comme 4, 6 ou 8.

```
public function movePieces(event:Event) {
    var madeMove:Boolean = false;
    for(var row:int=0;row<8;row++) {
        for(var col:int=0;col<8;col++) {
            if (grid[col][row] != null) {

                // Déplacement vers le bas
                if (grid[col][row].y <
                       grid[col][row].row*spacing+offsetY) {
                    grid[col][row].y += 5;
                    madeMove = true;

                // Déplacement vers le haut
                } else if (grid[col][row].y >
                       grid[col][row].row*spacing+offsetY) {
                    grid[col][row].y -= 5;
                    madeMove = true;

                // Déplacement vers la droite
```

```
                } else if (grid[col][row].x <
                        grid[col][row].col*spacing+offsetX) {
                    grid[col][row].x += 5;
                    madeMove = true;

                // Déplacement vers la gauche
                } else if (grid[col][row].x >
                        grid[col][row].col*spacing+offsetX) {
                    grid[col][row].x -= 5;
                    madeMove = true;
                }
            }
        }
    }
```

Au début de `movePieces`, nous positionnons la variable booléenne `madeMove` à `false`. Ensuite, lorsqu'une animation est requise, nous la positionnons à `true`. En d'autres termes, si `movePieces` ne fait rien, `madeMove` vaut `false`.

Ensuite, cette valeur est comparée aux propriétés `isDropping` et `isSwapping` de la classe. Si `isDropping` vaut `true` et `madeMove` vaut `false`, cela signifie que toutes les pièces qui tombaient ont atterri. Il est temps de rechercher d'autres correspondances.

En outre, si `isSwapping` vaut `true` et `madeMove` vaut `false`, cela signifie que deux pièces viennent juste de terminer leur permutation. Dans ce cas, il est également temps de rechercher des correspondances :

```
        // Si la chute des pièces est terminée
        if (isDropping && !madeMove) {
            isDropping = false;
            findAndRemoveMatches();

        // Si la permutation est terminée
        } else if (isSwapping && !madeMove) {
            isSwapping = false;
            findAndRemoveMatches();
        }
    }
```

Trouver des correspondances

Deux parties du programme Match Three posent un véritable défi. La première est celle qui consiste à trouver des correspondances dans la grille. Il s'agit, comme nous l'avons vu au Chapitre 1, d'un excellent exemple de la technique de programmation qui consiste à décomposer le problème en problèmes plus petits. L'opération qui consiste à trouver des correspondances de trois pièces consécutives ou plus dans la même grille n'a rien de trivial. Elle ne peut être résolue en une simple étape. Il ne faut donc pas considérer qu'il s'agisse d'un unique problème à résoudre.

Décomposer la tâche en étapes plus petites

Nous devons au lieu de cela décomposer cette tâche en problèmes plus petits et poursuivre cette décomposition jusqu'à ce que les problèmes deviennent suffisamment simples pour être facilement résolus.

`findAndRemoveMatches` commence donc par décomposer la tâche en deux étapes : trouver des correspondances et les supprimer. La suppression des pièces est en fait plutôt simple. Elle implique simplement de supprimer les objets `Piece` du `gameSprite`, de positionner l'emplacement de `grid` correspondant à `null` et d'attribuer des points au joueur.

Le nombre de points attribués dépend du nombre de pièces consécutives alignées. Trois pièces donnent `(3-1)*50` ou 100 points par pièce pour un total de 300 points. Quatre pièces donnent `(4-1)*50` soit 150 points par pièce pour un total de 600 points.

L'absence de certaines `Piece` implique cependant que celles au-dessus devront se voir indiquer qu'elles sont suspendues dans les airs et doivent tomber. Cette tâche n'est pas triviale non plus.

Nous avons donc deux tâches non triviales à gérer : rechercher des correspondances et indiquer aux pièces au-dessus de celles supprimées qu'elles doivent tomber. Nous déléguons ces deux tâches à d'autres fonctions : `lookForMatches` et `affectAbove`. Nous réaliserons les autres tâches simples directement ici dans la fonction `findAndRemoveMatches`.

La fonction *findAndRemoveMatches*

Nous parcourons en boucle les correspondances trouvées et les plaçons dans le tableau `matches`. Ensuite, nous attribuons des points pour chaque correspondance puis nous parcourons tous les objets `Piece` à supprimer et les supprimons.

La technique qui consiste à déléguer des tâches difficiles à de nouvelles fonctions que vous n'avez pas encore créées est appelée programmation de haut en bas. Au lieu de se soucier de la manière de trouver des correspondances, nous considérons simplement une fonction `lookForMatches` qui s'occupera de cette tâche. Nous construisons le programme de haut en bas, en s'occupant d'abord des considérations d'ordre général et en se souciant ensuite des fonctions qui s'occupent des points de détail.

```
// Récupérer les correspondances et les supprimer, attribuer les points
public function findAndRemoveMatches() {
    // Obtenir la liste des correspondances
    var matches:Array = lookForMatches();
    for(var i:int=0;i<matches.length;i++) {
```

```
            var numPoints:Number = (matches[i].length-1)*50;
            for(var j:int=0;j<matches[i].length;j++) {
                if (gameSprite.contains(matches[i][j])) {
                    var pb = new PointBurst(this,numPoints,matches[i][j].x,matches[i][j].y);
                    addScore(numPoints);
                    gameSprite.removeChild(matches[i][j]);
                    grid[matches[i][j].col][matches[i][j].row] = null;
                    affectAbove(matches[i][j]);
                }
            }
        }
```

La fonction `findAndRemoveMatches` a deux tâches de plus à réaliser. Tout d'abord, elle appelle `addNewPieces` pour remplacer tous les objets `Piece` manquants dans une colonne. Ensuite, elle appelle `lookForPossibles` pour s'assurer qu'il existe encore des déplacements possibles. Elle ne doit le faire que si aucune correspondance n'est trouvée. Cela ne se produit que si `findAndRemoveMatches` a été appelée après que les nouvelles pièces ont fini de tomber et qu'aucune correspondance n'est trouvée :

```
        // Ajouter les nouvelles pièces en haut de la grille
        addNewPieces();

        // Aucune correspondance trouvée, peut-être fin de partie ?
        if (matches.length == 0) {
            if (!lookForPossibles()) {
                endGame();
            }
        }
    }
```

La fonction *lookForMatches*

La fonction `lookForMatches` a toujours une tâche assez gigantesque à gérer. Elle doit créer un tableau de toutes les correspondances trouvées. Elle doit retrouver des correspondances horizontales et verticales de plus de deux pièces. Elle le fait en parcourant en boucle les lignes pour commencer, puis les colonnes. Elle ne doit vérifier que les six premiers emplacements dans chaque ligne et chaque colonne, car une correspondance qui commence au septième emplacement ne peut faire que deux de longueur et que le huitième emplacement n'est suivi d'aucune pièce.

Les fonctions `getMatchHoriz` et `getMatchVert` se chargent de la tâche déléguée qui consiste à déterminer la longueur d'une correspondance à un emplacement donné de la grille. Par exemple, si l'emplacement 3,6 correspond à une `Piece` de type 4, si l'emplacement 4,6 est aussi de type 4, mais que 5,6 soit de type 1, l'appel `getMatchHoriz(3,6)` doit retourner 2, car l'emplacement 3,6 démarre une série de deux pièces correspondantes.

Si une série est trouvée, nous souhaitons également prolonger la boucle de quelques étapes. S'il y a un alignement de quatre pièces identiques aux positions 2,1, 2,2, 2,3 et 2,4, nous vérifions simplement 2,1 et obtenons le résultat 4, puis sautons 2,2 2,3 et 2,4 afin d'examiner directement 2,5.

À chaque fois que `getMatchHoriz` ou `getMatchVert` retrouvent une correspondance, elles retournent un tableau contenant chacun des objets `Piece` dans la correspondance. Ces tableaux sont ensuite ajoutés au tableau `matches` dans `lookForMatches`, qui est à son tour retourné au code ayant appelé `lookForMatches` :

```
//Retourner un tableau de toutes les correspondances trouvées
public function lookForMatches():Array {
    var matchList:Array = new Array();

    // Rechercher les correspondances horizontales
    for (var row:int=0;row<8;row++) {
        for(var col:int=0;col<6;col++) {
            var match:Array = getMatchHoriz(col,row);
            if (match.length > 2) {
                matchList.push(match);
                col += match.length-1;
            }
        }
    }
    // Rechercher les correspondances verticales
    for(col=0;col<8;col++) {
        for (row=0;row<6;row++) {
            match = getMatchVert(col,row);
            if (match.length > 2) {
                matchList.push(match);
                row += match.length-1;
            }

        }
    }
    return matchList;
}
```

Les fonctions *getMatchHoriz* et *getMatchVert*

La fonction `getMatchHoriz` a maintenant une étape spécialisée à réaliser. Avec une colonne et une ligne, elle vérifie la `Piece` suivante afin de voir si son type correspond. Si c'est le cas, elle est ajoutée à un tableau. La fonction continue d'avancer horizontalement jusqu'à trouver une `Piece` qui ne correspond pas. Ensuite, elle retourne le tableau qu'elle a compilé. Ce tableau peut en fin de compte ne contenir qu'une seule `Piece` (celle de la colonne et de la ligne d'origine) si la suivante ne correspond pas.

En revanche, si la suivante correspond et celle d'après aussi, elle retourne une série de trois `Piece` :

```
// Rechercher des correspondances horizontales à partir de ce point
public function getMatchHoriz(col,row):Array {
    var match:Array = new Array(grid[col][row]);
    for(var i:int=1;col+i<8;i++) {
        if (grid[col][row].type == grid[col+i][row].type) {
            match.push(grid[col+i][row]);
        } else {
            return match;
        }
    }
    return match;
}
```

La fonction `getMatchVert` est presque identique à la fonction `getMatchHoriz`, à ceci près qu'elle effectue sa recherche dans les colonnes et non dans les lignes :

```
// Rechercher des correspondances verticales à partir de ce point
public function getMatchVert(col,row):Array {
    var match:Array = new Array(grid[col][row]);
    for(var i:int=1;row+i<8;i++) {
        if (grid[col][row].type == grid[col][row+i].type) {
            match.push(grid[col][row+i]);
        } else {
            return match;
        }
    }
    return match;
}
```

La fonction *affectAbove*

Nous allons continuer de travailler afin de construire pour `findAndRemoveMatches` toutes les fonctions dont elle a besoin. La suivante est `affectAbove`. Nous lui passons un objet `Piece` et attendons d'elle qu'elle indique à tous les objets `Piece` au-dessus qu'ils doivent descendre d'un cran. Il s'agit en effet d'une `Piece` qui dit : "Je m'en vais, aussi venez remplir l'espace laissé libre."

Une boucle examine dans la colonne les pièces qui se trouvent au-dessus de la pièce actuelle. Si la pièce actuelle est 5,6, elle examine donc dans l'ordre 5,5, 5,4, 5,3, 5,2, 5,1 et 5,0. La ligne (`row`) de ces pièces est alors incrémentée d'une unité. En outre, la `Piece` indique à `grid` qu'elle se trouve dans un nouvel emplacement.

Rappelez-vous qu'avec `movePieces` nous n'avons pas à nous soucier de la manière dont une `Piece` doit s'animer pour parvenir à un nouvel emplacement.

Il nous suffit de changer les propriétés `col` ou `row` pour que la fonction s'en charge par elle-même :

```
// Inviter toutes les pièces au-dessus de celle-ci à descendre d'un cran
public function affectAbove(piece:Piece) {
    for(var row:int=piece.row-1;row>=0;row--) {
        if (grid[piece.col][row] != null) {
            grid[piece.col][row].row++;
            grid[piece.col][row+1] = grid[piece.col][row];
            grid[piece.col][row] = null;
        }
    }
}
```

La fonction *addNewPieces*

La prochaine fonction que nous devons créer est `addNewPieces`. Elle examine chaque colonne, puis chaque emplacement de la grille pour chaque colonne et compte le nombre d'emplacements positionnés à `null`. Pour chacun d'entre eux, un nouvel objet `Piece` est ajouté. Si la valeur `col` et `row` est définie de manière à correspondre à sa destination finale, la valeur `y` est en revanche définie pour correspondre à la ligne au-dessus de la ligne supérieure, afin qu'elle semble tomber de plus haut. En outre, la variable booléenne `isDropping` se voit attribuer la valeur `true` pour indiquer que l'animation est en cours :

```
// S'il manque des pièces dans une colonne, en ajouter pour les faire tomber
public function addNewPieces() {
    for(var col:int=0;col<8;col++) {
        var missingPieces:int = 0;
        for(var row:int=7;row>=0;row--) {
            if (grid[col][row] == null) {
                var newPiece:Piece = addPiece(col,row);
                newPiece.y = offsetY-spacing-spacing*missingPieces++;
                isDropping = true;
            }
        }
    }
}
```

Trouver des déplacements possibles

S'il est compliqué de trouver des correspondances, il est en revanche plus simple de trouver des correspondances possibles. Il s'agit cette fois non plus d'alignements de trois pièces mais de possibilités de tels alignements.

La réponse la plus simple consiste à balayer la grille entière en opérant toutes les permutations : 0,0 avec 1,0, puis 1,0 avec 2,0, etc. À chaque permutation, on vérifie les correspondances. Dès qu'une permutation conduit à obtenir une correspondance valide, nous cessons la recherche et retournons `true`.

Cette approche dite par "force brute" fonctionnerait, mais elle risquerait d'être assez lente, notamment sur les ordinateurs plus anciens. Il existe une technique plus efficace.

Si vous considérez ce qu'il faut pour faire une correspondance, vous verrez que certains motifs se constituent. En général, vous avez deux `Piece` du même type dans une ligne. L'emplacement à côté de ces deux `Piece` est d'un type différent mais peut être permuté dans trois sens afin d'y placer une autre `Piece` qui pourrait correspondre. Sans cela, vous avez deux `Piece` espacées d'un cran l'une de l'autre et une permutation pourrait insérer une `Piece` correspondante entre les deux.

La Figure 8.9 présente ces deux configurations en les décomposant en six motifs possibles au final. Horizontalement, la `Piece` manquante dans la correspondance peut venir de la gauche ou de la droite alors que, verticalement, elle peut venir du haut ou du bas.

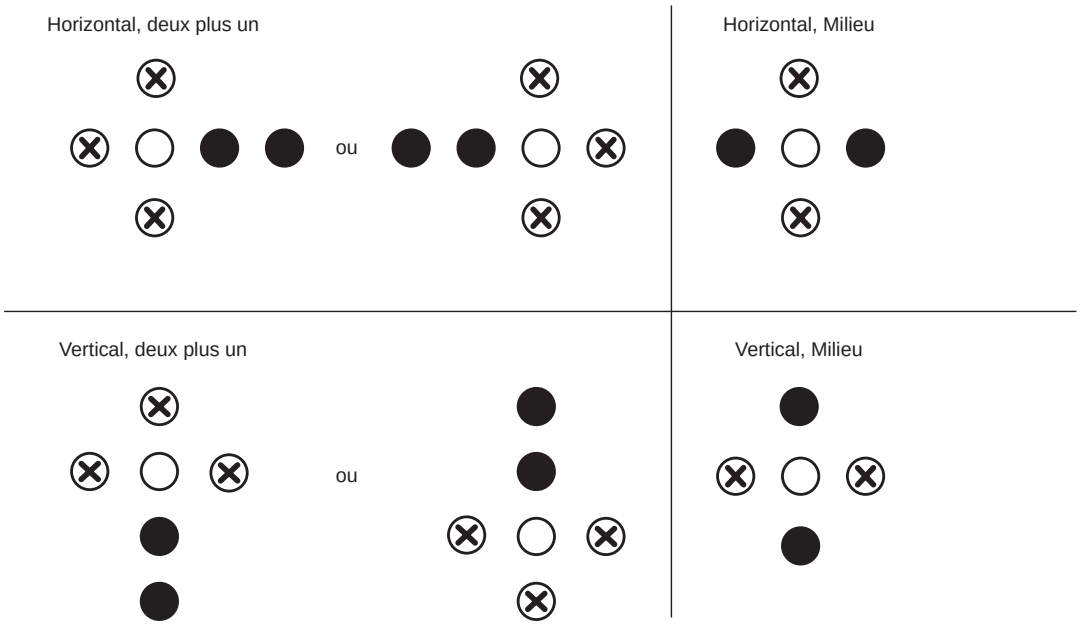


Figure 8.9
Les cercles remplis représentent les pièces qui restent en place. Les cercles vides représentent l'espace qui doit être rempli par la pièce concordante. Les cercles marqués d'une croix désignent les emplacements possibles d'où pourrait provenir cette pièce concordante.

Sachant qu'il n'y a que quelques motifs à examiner, nous pouvons écrire une fonction qui récupère une liste des emplacements et détermine si le motif concorde. Conformément à la technique de programmation de haut en bas, nous pouvons commencer par écrire `lookForPossibles` puis nous soucier par la suite de la fonction de correspondance des motifs.

Si l'on examine le premier motif à la Figure 8.9, on remarque que, si deux emplacements contiennent une pièce de même type, nous obtenons un résultat positif dès lors que l'un parmi trois autres contient une pièce correspondante. Si le cercle rempli de gauche est considéré être le point `0,0`, le suivant (`1,0`) doit correspondre et il doit y avoir au moins une `Piece` correspondante aux emplacements `-1,-1`, `-2,0` ou `-1,1`. La correspondance peut aussi se trouver du côté droit de la paire initiale, au niveau des positions `2,-1`, `2,1` et `3,0`.

Récapitulons : il y a une `Piece` de départ. Il y a ensuite une unique position qui doit correspondre à la `Piece` de départ. Enfin, il y a six autres positions dans lesquelles au moins une `Piece` doit correspondre. La Figure 8.10 présente cette combinatoire sous forme de diagramme.

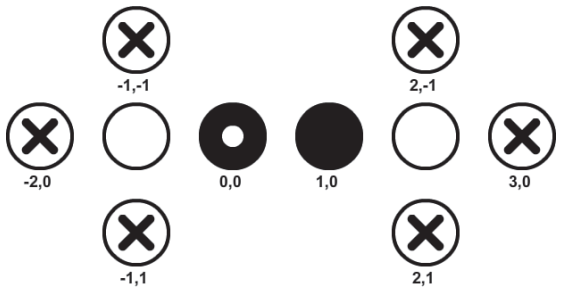


Figure 8.10
La position `1,0` doit correspondre à `0,0`. Au moins un des six emplacements X doit ensuite aussi correspondre `0,0`.

L'appel de fonction passera un tableau des positions qui doivent correspondre et un second tableau des positions où au moins une correspondance doit être trouvée. Il ressemblera ainsi à ceci :

```
matchPattern(col, row,  [[1,0]],  [[-2,0],[-1,-1],[-1,1],[2,-1],[2,1],[3,0]]))
```

Nous avons besoin d'un appel de fonction similaire pour gérer le cas "Horizontal, milieu" présenté à la Figure 8.9. Idem ensuite pour les motifs verticaux. La fonction `lookForPossibles` les recherche tous, à toutes les positions dans la grille :

```
// Vérifier si un déplacement est possible dans la grille
public function lookForPossibles() {
    for(var col:int=0;col<8;col++) {
        for(var  row:int=0;row<8;row++) {

            // Possibilité horizontale, deux plus un
            if (matchPattern(col, row,
                   [[1,0]], [[-2,0],[-1,-1],[-1,1],[2,-1],[2,1],[3,0]])) {
                return true;
            }

            // Possibilité horizontale, milieu
            if (matchPattern(col, row, [[2,0]], [[1,-1],[1,1]])) {
                return true;
            }
```

```
                // Possibilité verticale, deux plus un
                if (matchPattern(col, row,
                       [[0,1]], [[0,-2],[-1,-1],[1,-1],[-1,2],[1,2],[0,3]])) {
                    return true;
                }

                // Possibilité verticale, milieu
                if (matchPattern(col, row, [[0,2]], [[-1,1],[1,1]])) {
                    return true;
                }
            }
        }

        // Aucun déplacement possible n'a été trouvé
        return false;
    }
```

La fonction `matchPattern`, quoiqu'elle possède une tâche importante à réaliser, n'est pas une fonction très longue. Elle doit obtenir le type de la `Piece` à la position de colonne et de ligne spécifiée. Ensuite, elle examine la liste `mustHave` et vérifie la `Piece` à la position relative. Si elle ne correspond pas, inutile de continuer et la fonction retourne `false`.

Sinon chacune des `Piece` dans `needOne` est vérifiée. Si l'une quelconque correspond, la fonction retourne `true`. Si aucune ne correspond, la fonction s'arrête et retourne `false` :

```
    public function matchPattern(col,row:uint, mustHave, needOne:Array) {
        var thisType:int = grid[col][row].type;

        // S'assurer que les pièces impératives sont là
        for(var i:int=0;i<mustHave.length;i++) {
            if (!matchType(col+mustHave[i][0],
                   row+mustHave[i][1], thisType)) {
                return false;
            }

        }

        // S'assurer qu'il existe au moins une des pièces requises
        for(i=0;i<needOne.length;i++) {
            if (matchType(col+needOne[i][0],
                   row+needOne[i][1], thisType)) {
                return true;
            }
        }
        return false;
    }
```

Toutes les comparaisons dans `matchPattern` sont opérées *via* des appels à `matchType`. C'est que nous essayons souvent d'examiner des `Piece` qui ne se trouvent pas dans la grille. Par exemple, si la colonne et la ligne passées à `matchPattern` sont `5,0` et que la `Piece` décalée de `-1,-1` est examinée, nous examinons en fait `grid[4,-1]`, qui n'est pas définie, car il n'existe pas d'élément `-1` dans un tableau.

La fonction `matchType` recherche des valeurs d'emplacement de la grille qui se trouvent hors de la grille que nous avons définie et retourne instantanément `false` lorsque cela se produit. Sinon la valeur de `grid` est examinée et la valeur `true` est retournée si le type correspond :

```
public function matchType(col,row,type:int) {
            // S'assurer que col et row ne sont pas hors-limites
    if ((col < 0) || (col > 7) || (row < 0) || (row > 7)) return false;
    return (grid[col][row].type == type);
}
```

Mémorisation du score et fin de partie

Dans `findAndRemoveMatches`, nous avions appelé `addScore` afin d'attribuer des points au joueur. Cette fonction simple ajoute des points au score du joueur et met à jour le champ texte à l'écran :

```
public function addScore(numPoints:int) {
    gameScore += numPoints;
    MovieClip(root).scoreDisplay.text = String(gameScore);
}
```

Lorsqu'il ne reste plus de correspondance possible, la fonction `endGame` conduit le scénario principal à l'image `gameover`. Elle utilise aussi `swapChildIndex` pour placer le `gameSprite` à l'arrière, afin que les sprites de l'image `gameover` se trouvent au-dessus de la grille du jeu.

Nous devons procéder ainsi parce que nous ne supprimerons pas la grille du jeu à la fin de la partie. Au lieu de cela, nous la conserverons à cet endroit pour que le joueur puisse l'examiner :

```
public function endGame() {
    // Ramener à l'arrière
    setChildIndex(gameSprite,0);
    // Aller à la fin de partie
    gotoAndStop("gameover");
}
```

Nous nous débarrassons de la grille (`grid`) et du sprite `gameSprite` lorsque le joueur est prêt à passer à la suite. La fonction `cleanUp` s'en charge :

```
public function cleanUp() {
    grid = null;
    removeChild(gameSprite);
    gameSprite = null;
    removeEventListener(Event.ENTER_FRAME,movePieces);
}
```

Dans le scénario principal, la fonction liée au bouton Play Again appelle `cleanUp` juste avant de revenir à l'image précédente afin de commencer une nouvelle partie.

Modifier le jeu

L'une des décisions importantes à prendre est de savoir si vous souhaitez six ou sept variantes de pièces dans le jeu. La plupart des jeux Match Three en comptent six. J'en ai utilisé sept par le passé, et cela fonctionnait également. Le recours à sept pièces accélère la fin du jeu.

Les points de bonus font partie des améliorations importantes qui peuvent être effectuées. Un calque de graphisme supplémentaire peut être ajouté au clip `Piece`, de manière analogue à la bordure de sélection. Il peut être rendu visible sur des pièces aléatoires afin d'indiquer des points de bonus. Une propriété `bonus` peut être ajoutée à la `Piece` également, qui pourrait déclencher un second appel à `addScore` lorsque cette `Piece` est supprimée.

Les indications peuvent être un moyen de rendre le jeu plus attrayant pour le joueur. Lorsque `lookForPossibles` est appelée, elle appelle `matchType` un certain nombre de fois. Si une correspondance possible est trouvée dans la seconde boucle à l'intérieur de `matchType`, la valeur `true` est retournée. La position précise que matchType examine à ce stade est une `Piece` qui peut être utilisée dans une permutation pour créer un alignement gagnant. Elle peut être placée dans une nouvelle variable appelée par exemple `hintLocation` et cet emplacement peut être utilisé pour mettre en valeur une pièce lorsque le joueur clique sur le bouton Indice.

9

Jeux de mots : pendu et mots mêlés

Au sommaire de ce chapitre :

- Chaînes et champs texte
- Pendu
- Mots mêlés

L'utilisation des lettres et des mots pour les jeux s'est considérablement popularisée depuis le milieu du XX^e siècle avec les jeux de société tels que le Scrabble ou les jeux sur papier comme les mots croisés et les mots mêlés.

Ces jeux fonctionnent bien comme jeux d'ordinateur et sur le Web. Dans ce chapitre, nous en étudierons deux versions classiques : le jeu du pendu et le jeu des mots mêlés. Avant cela, nous devons cependant voir comment ActionScript gère les chaînes et les champs texte.

Chaînes et champs texte

Codes sources

http://flashgameu.com

A3GPU09_TextExamples.zip

Avant d'essayer de créer des jeux de mots, il est utile de compendre comment ActionScript 3.0 gère les chaînes et les champs texte, car nous nous en servirons fréquemment dans nos jeux.

Gestion des chaînes en ActionScript 3.0

En ActionScript, une variable `String` (chaîne) est une séquence de caractères. Nous avons déjà utilisé des chaînes dans ce livre, sans vraiment nous soucier de la manière de réaliser des opérations avancées avec elles.

Pour créer une chaîne, il suffit d'attribuer des caractères entourés de guillemets à une variable de type String :

```
var myString:String = "Why is a raven like a writing desk?";
```

Déconstruction des chaînes

Les chaînes peuvent être déconstruites avec une variété de fonctions. Pour obtenir un unique caractère situé à un emplacement particulier, vous pouvez utiliser `charAt` :

```
myString.charAt(9)
```

Le résultat est ici `"r"`.

ActionScript commence à compter les positions des caractères dans les chaînes à partir du caractère 0. Le caractère 0 dans l'exemple précédent est ainsi "W", de sorte que le caractère 9 est "r".

Il est également possible d'utiliser `substr` pour obtenir un ou plusieurs caractères de la chaîne. Le premier paramètre est la position de départ et le second, le nombre de caractères à retourner :

```
myString.substr(9,5)
```

Cette instruction retourne `"raven"`.

La fonction `substring` est une variante qui prend en paramètre la position de départ et celle de fin. Elle retourne la portion de chaîne allant du caractère de la position de départ jusqu'à un caractère avant la position de fin :

```
myString.substring(9,14)
```

Cette instruction retourne `"raven"`.

La fonction `slice` agit comme la fonction `substring`, sauf quant à l'interprétation du second paramètre. Avec `substring`, les paramètres sont inversés si le second est inférieur au premier. `myString.substring(9,14)` équivaut donc à `myString.substring(14,9)`.

La fonction `slice` vous permet d'utiliser des valeurs négatives pour le second paramètre. Elle compte en arrière à partir de la fin de la chaîne. Ainsi, `myString.slice(9,-21)` retourne `"raven"`.

`substring` et `slice` vous permettent toutes deux d'omettre le second paramètre pour obtenir le reste de la chaîne.

```
myString.slice(9)
```

Cette instruction retourne `"raven like a writing desk?"`.

Comparer et rechercher des chaînes

Pour comparer deux chaînes, il suffit d'utiliser l'opérateur `==` :

```
var testString = "raven";
trace(testString == "raven");
```

Ce code retourne `true`. L'opérateur est cependant sensible à la casse, de sorte que le code suivant retourne `false` :

```
trace(testString == "Raven");
```

Si vous souhaitez comparer deux chaînes sans tenir compte de la casse des caractères, convertissez l'une ou les deux en majuscules ou en minuscules. C'est ce que permettent de faire les fonctions `toUpperCase` et `toLowerCase` :

```
testString.toLowerCase() == "Raven".toLowerCase()
```

Pour trouver une chaîne à l'intérieur d'une autre chaîne, utilisez `indexOf` :

```
myString.indexOf("raven")
```

Cette instruction retourne `9`. Vous pouvez également utiliser `lastIndexOf` pour trouver la dernière occurrence d'une chaîne à l'intérieur d'une autre chaîne :

```
myString.indexOf("a")
myString.lastIndexOf("a")
```

La première ligne retourne `7` et la seconde, `20`. Ces valeurs correspondent à la première et à la dernière position de la lettre dans la chaîne `"Why is a raven like a writing desk?"`.

Vous pouvez également fournir un second paramètre à `indexOf` et à `lastIndexOf`. Ce nombre indique où commencer la recherche dans la chaîne au lieu de démarrer au début.

La plupart du temps, lorsque vous utiliserez `indexOf`, vous chercherez non pas à connaître la position de la chaîne mais à savoir si elle existe. Si c'est le cas, `indexOf` retourne un nombre supérieur ou égal à 0. Si ce n'est pas le cas, elle retourne `-1`. Vous pouvez donc déterminer si une chaîne se trouve à l'intérieur d'une autre en utilisant une instruction comme la suivante :

```
(myString.indexOf("raven") != -1)
```

La fonction `search` offre un autre moyen de trouver une chaîne à l'intérieur d'une autre chaîne :

```
myString.search("raven")
```

Cette instruction retourne `9`.

Comme indiqué précédemment, la fonction `search` peut prendre une chaîne en paramètre, mais elle peut également prendre ce que l'on appelle une expression régulière.

Une expression régulière est un motif utilisé pour trouver ou remplacer des chaînes à l'intérieur d'autres chaînes. Les expressions régulières sont utilisées dans de nombreux langages de programmation et outils logiciels.

Le sujet des expressions régulières est trop vaste pour être traité ici. Il existe d'ailleurs plusieurs livres de plus de mille pages qui lui sont entièrement consacrés ! De nombreux sites Web en traitent en détail. Consultez le site ***http://flashgameu.com*** *pour trouver des liens sur ce sujet.*

```
myString.search(/raven/);
```

Cet exemple correspond au type le plus simple d'expression régulière et équivaut à l'instruction précédente utilisant `search`. Vous remarquerez que c'est le caractère `/` qui est utilisé et non les guillemets pour entourer les caractères.

Vous pouvez également inclure des options après la barre oblique dans l'expression régulière. La plus utile serait ici le `i`, pour ignorer la casse des caractères :

```
myString.search(/Raven/i);
```

Cet exemple retourne `9`, bien qu'il utilise un R majuscule.

Vous pouvez aussi utiliser des jokers dans les expressions régulières. Par exemple, le point représente n'importe quel caractère :

```
myString.search(/r...n/)
```

Cet exemple retourne `9` parce que le mot raven correspond au motif du r suivi par trois caractères quelconques, puis un n.

```
myString.search(/r.*n/)
```

Cet exemple retourne aussi 9, le motif désignant un r suivi par n'importe quel nombre de caractères, suivi par un n.

Construire et modifier des chaînes

Vous pouvez ajouter des caractères à une chaîne en utilisant l'opérateur `+`. ActionScript déterminera qu'il s'agit d'un objet `String` (une chaîne) et non d'un nombre et ajoutera les caractères au lieu de les additionner. Vous pouvez également utiliser `+=` pour réaliser un simple ajout :

```
myString = "Why is a raven like";
myString += " a writing desk?";
```

Pour placer quelque chose avant une chaîne existante, utilisez du code comme le suivant :

```
myString = "a writing desk?";
myString = "Why is a raven like "+myString;
```

Alors que la fonction `search` recherche et retourne une valeur d'index, la fonction `replace` prend une expression régulière et l'utilise pour remplacer une portion de la chaîne :

```
myString.replace("raven","door mouse")
```

Cet exemple retourne `"Why is a door mouse like a writing desk?"`

Vous pouvez également utiliser une expression régulière dans le premier paramètre. Cette approche permet de réaliser des opérations très complexes, par exemple pour déplacer une portion de texte à l'intérieur d'un texte au lieu d'importer un texte de remplacement :

```
myString.replace(/(raven)(.*)(writing desk)/g,"$3$2$1")
```

Cet exemple de code recherche `raven` et `writing desk` dans la chaîne, séparés par n'importe quel nombre de caractères. Il réordonne ensuite la chaîne, en plaçant d'abord `writing desk`, en dernier `raven` et les mêmes caractères entre les deux.

Conversions entre chaînes et tableaux

Les chaînes et les tableaux sont aussi utiles les uns que les autres pour stocker des listes d'informations. Il est donc intéressant de pouvoir effectuer des conversions entre les deux.

Par exemple, vous pourriez souhaiter créer un tableau à partir d'une chaîne comme `"apple,orange,banana"`. Pour cela, vous pouvez utiliser la commande `split` :

```
var myList:String = "apple,orange,banana";
var myArray:Array = myList.split(",");
```

Vous pouvez inverser le processus est utilisant la commande `join` :

```
var myList:String = myArray.join(",");
```

Dans les deux cas, le caractère passé dans la fonction représente le caractère utilisé pour séparer les éléments dans la chaîne. Si vous utilisez la commande `join`, la chaîne `string` résultante est recollée en insérant des virgules entre les éléments.

Synthèse des fonctions de chaîne

Le Tableau 9.1 contient toutes les fonctions de chaîne dont nous avons traité et quelques autres.

Tableau 9.1 : Fonctions de chaîne

Fonction	*Syntaxe*	*Description*
`charAt`	`myString.charAt(`*`pos`*`)`	Retourne le caractère à l'emplacement indiqué
`charCodeAt`	`String.charCodeAt(`*`pos`*`)`	Retourne le code de caractère du caractère à l'emplacement indiqué
`concat`	`myString.concat(`*`autreChaine`*`)`	Retourne une nouvelle chaîne avec la seconde chaîne ajoutée à la première
`fromCharCode`	`String.fromCharCode(`*`num`*`)`	Retourne le caractère correspondant au code de caractère
`indexOf`	`myString.indexOf (`*`chaîneIntérieure,posDépart`*`)`	Retourne l'emplacement de la chaîne interne dans la chaîne principale
`join`	`myArray.join(`*`car`*`)`	Combine les éléments dans un tableau afin de créer une chaîne
`lastIndexOf`	`myString.lastIndexOf (`*`chaîneIntérieure,posDépart`*`)`	Retourne le dernier emplacement de la chaîne intérieure dans la chaîne principale
`match`	`myString.match(`*`expression`*`)`	Retourne la sous-chaîne qui correspond au motif
`replace`	`myString.replace(`*`expression, remplacement`*`)`	Remplace le motif

Tableau 9.1 : Fonctions de chaîne *(Suite)*

Fonction	*Syntaxe*	*Description*
`search`	`myString.search(`*`expression`*`)`	Trouve l'emplacement de la sous-chaîne correspondant au motif
`slice`	`myString.slice(`*`début,fin`*`)`	Retourne la sous-chaîne
`split`	`myString.split(`*`car`*`)`	Décompose la chaîne dans un tableau
`string`	`String(`*`pasUneChaîne`*`)`	Convertit un nombre ou une autre valeur en une chaîne
`substr`	`myString.substr(`*`début,long`*`)`	Retourne la sous-chaîne
`substring`	`myString.substr(`*`début,fin`*`)`	Retourne la sous-chaîne
`toLowerCase`	`myString.toLowerCase()`	Retourne la chaîne en lettres minuscules
`toUpperCase`	`myString.toUpperCase()`	Retourne la chaîne en lettres majuscules

Appliquer une mise en forme au champ texte

Pour placer du texte à l'écran, vous devez créer un nouvel objet `TextField`, autrement dit un champ texte. Nous avons utilisé ces champs dans les précédents chapitres afin de créer des messages texte et des affichages de score.

Si vous souhaitez utiliser autre chose que la police et le style par défaut, vous devez également créer un objet `TextFormat` et l'attribuer au champ texte. Et, pour l'utilisation avancée du texte dans les jeux, nous devrons encore inclure des polices dans nos animations.

L'objet *TextFormat*

L'objet `TextFormat` se crée généralement juste avant de créer un objet `TextField`. Il peut aussi l'être au début d'une classe si vous savez que vous allez utiliser ce format pour plusieurs des champs texte que vous allez créer.

`TextFormat` n'est en fait rien d'autre qu'un conteneur pour un ensemble de propriétés qui contrôlent l'apparence du texte.

ActionScript vous permet également de créer des feuilles de style comparables aux CSS utilisées dans les documents HTML. Ces feuilles de style ne sont cependant utiles que pour les champs texte formatés en HTML. Pour notre part, nous n'utiliserons que des champs texte bruts dans nos jeux.

Vous avez deux choix lors de la création d'un `TextFormat`. Le premier consiste à créer simplement un objet `TextFormat` vide puis à définir chacune de ses propriétés. Le second consiste à définir plusieurs des propriétés les plus courantes dans la déclaration `TextFormat`.

Voici un exemple de la méthode de création rapide d'un objet `TextFormat` :

```
var letterFormat:TextFormat = new
    TextFormat("Monaco",36,0x000000,true,false,false,null,null,"center");
```

Il est évidemment important de se rappeler l'ordre exact des paramètres pour `TextFormat`. Il est le suivant : police, taille, couleur, gras, italique, souligné, url, cible et alignement. Vous pouvez inclure le nombre de propriétés que vous souhaitez, pourvu qu'elles soient indiquées dans cet ordre. Utilisez `null` pour passer les propriétés que vous ne souhaitez pas définir.

La liste des paramètres est en fait plus étendue, mais je les ai omis dans le précédent exemple : marge gauche (`leftMargin`), marge droite (`rightMargin`), retrait (`indent`) et interlignage (`leading`).

Voici maintenant la méthode longue :

```
var letterFormat:TextFormat = new TextFormat();
letterFormat.font = "Monaco";
letterFormat.size = 36;
letterFormat.color = 0x000000;
letterFormat.bold = true;
letterFormat.align = "center";
```

Vous remarquerez que j'ai laissé les propriétés `italic` et `underline`, car la valeur `false` est utilisée par défaut pour les deux.

Le Tableau 9.2 présente une synthèse de toutes les propriétés `TextFormat`.

Tableau 9.2 : Propriétés de TextFormat

Propriété	*Valeurs d'exemple*	*Description*
`align`	`TextFormatAlign.LEFT`, `TextFormatAlign.RIGHT`, `TextFormatAlign.CENTER`, `TextFormatALign.JUSTIFY`	Alignement du texte
`blockIndent`	Nombre	Retrait de toutes les lignes d'un paragraphe
`bold`	`true/false`	Passe le texte en gras
`bullet`	`true/false`	Affiche le texte sous forme de liste à puces

Tableau 9.2 : Propriétés de TextFormat *(Suite)*

Propriété	*Valeurs d'exemple*	*Description*
`color`	Couleur	Couleur du texte (par exemple, 0x000000)
`font`	Nom de police	Police à utiliser
`indent`	Nombre	Retrait de la première ligne du paragraphe uniquement
`italic`	`true/false`	Passe le texte en italique
`kerning`	`true/false`	Active l'espace spécial de certains caractères (crénage) dans certaines polices
`leading`	Nombre	Espacement vertical entre les lignes (interlignage)
`leftMargin`	Nombre	Espace supplémentaire à gauche
`letterSpacing`	Nombre	Espace supplémentaire entre les caractères
`rightMargin`	Nombre	Espace supplémentaire à droite
`size`	Nombre	Taille de la police
`tabStops`	Tableau de nombres	Emplacements de tabulation définis
`target`	Chaîne	Cible navigateur d'un lien (par exemple, `"_blank"`)
`underline`	`true/false`	Souligne le texte
`url`	Chaîne	L'URL du lien

Créer des objets *TextField*

Une fois que vous avez un format, il vous faut un champ texte auquel l'appliquer. L'objet `TextField` se crée à la manière d'un `Sprite`. Il s'agit en fait dans les deux cas d'un objet d'affichage. Tous deux peuvent être ajoutés à d'autres sprites ou à d'autres clips avec `addChild` :

```
var myTextField:TextField = new TextField();
addChild(myTextField);
```

Pour attribuer un format à un champ, la meilleure méthode consiste à utiliser la propriété `defaultTextFormat` :

```
myTextField.defaultTextFormat = letterFormat;
```

L'autre possibilité consiste à utiliser la fonction `setTextFormat`. Le problème tient alors à ce que, lorsque vous définissez la propriété `text` du champ, la mise en forme du texte revient à la mise en forme par défaut pour ce champ :

```
myTextField.setTextFormat(letterFormat);
```

L'avantage de `setTextFormat` tient à ce que vous pouvez ajouter un deuxième et un troisième paramètre pour spécifier les caractères de début et de fin pour la mise en forme. Vous pouvez formater un fragment de texte au lieu du texte entier.

Dans les jeux, nous utilisons généralement de petits champs texte pour différents usages comme l'affichage du score, du niveau, du temps de jeu, du nombre de vies, et ainsi de suite. Ces champs ne requièrent pas plusieurs formats de texte et sont souvent mis à jour. La définition de `defaultTextFormat` constitue donc la meilleure approche dans la plupart des cas.

Après `defaultTextFormat`, la prochaine propriété la plus important pour nous est `selectable`. La plupart des champs texte que nous utiliserons pour les jeux ne sont destinés qu'à un rôle d'affichage uniquement où l'utilisateur n'est pas censé pouvoir cliquer dessus. Nous désactiverons ainsi la propriété `selectable` afin que le curseur ne se modifie pas lorsqu'il survole le champ et que l'utilisateur ne puisse sélectionner le texte.

La propriété `border` du champ texte est un moyen utile de vérifier la taille et l'emplacement du champ créé avec ActionScript. Par exemple, si vous ne placez qu'un mot ou une lettre dans un champ, vous ne pourrez pas voir véritablement la taille du champ sans positionner `border` à `true`, ne serait-ce que temporairement, pour les besoins de votre test.

Le Tableau 9.3 présente certaines propriétés utiles de l'objet `TextField`.

Tableau 9.3 : Propriétés de TextField

Propriété	*Valeurs*	*Description*
`autoSize`	`TextFieldAutoSize.LEFT`, `TextFieldAutoSize.RIGHT`, `TextFieldAutoSize.CENTER`, `TextFieldAutoSize.NONE`	Redimensionne le champ texte afin de l'ajuster au texte placé dedans
`background`	`true/false`	Indique s'il y a un remplissage d'arrière-plan
`backgroundColor`	Couleur	Couleur du remplissage d'arrière-plan (par exemple, 0x000000)
`border`	`true/false`	Indique s'il y a une bordure
`borderColor`	Couleur	Couleur de la bordure (par exemple, 0x000000)
`defaultTextFormat`	Objet `TextFormat`	Définit le format de texte par défaut utilisé lorsque du nouveau texte est appliqué
`embedFonts`	`true/false`	Doit être positionné à true pour utiliser des polices incorporées

Tableau 9.3 : Propriétés de TextField *(Suite)*

Propriété	*Valeurs*	*Description*
`multiline`	`true/false`	Doit être positionné à true pour contenir plusieurs lignes de texte
`selectable`	`true/false`	Si true, l'utilisateur peut sélectionner le texte dans le champ texte
`text`	Chaîne	Définit le contenu entier du texte du champ
`textColor`	Couleur	Définit la couleur du texte (par exemple, 0x000000)
`type`	`TextFieldType.DYNAMIC`, `TextFieldType.INPUT`	Définit si l'utilisateur peut éditer le texte
`wordWrap`	`true/false`	Définit le renvoi à la ligne du texte

Polices

Si vous créez un jeu rapide en guise d'exemple, pour amuser vos amis ou simplement pour illustrer un point, vous pouvez vous en tenir aux polices de base. C'est ce que nous avons fait dans la plupart des jeux de ce livre, afin qu'ils restent aussi simples que possible.

Si vous développez un jeu pour un client ou pour votre site Web, vous devrez en revanche importer les polices que vous utilisez dans votre bibliothèque. Vous rendrez ainsi votre jeu indépendant des polices que les utilisateurs possèdent sur leur ordinateur. Cela vous permettra également d'utiliser des effets plus évolués avec les polices, comme la rotation et la transparence alpha.

Pour importer une police, accédez à la bibliothèque et choisissez Nouvelle police dans le menu déroulant (comme vous l'avez fait précédemment au Chapitre 7).

Après avoir importé la police, nommez-la, cliquez du bouton droit sous Windows ou avec la touche Ctrl enfoncée sur Mac pour sélectionner l'option Propriétés de liaison et incluez la police dans l'animation.

Les programmeurs oublient couramment de définir les Propriétés de liaison pour les polices. À la différence des clips, les propriétés de liaison n'apparaissent pas dans la boîte de dialogue Propriétés standard des polices. Elles sont difficiles à trouver et donc faciles à oublier. Aucune erreur ni aucun avertissement n'apparaît lorsque vous essayez d'utiliser des polices que vous avez oublié de lier.

Même après avoir incorporé certaines polices, vos champs texte n'utiliseront vos polices que si vous positionnez la propriété `embedFonts` à `true`.

En utilisant les polices qui se trouvent dans votre bibliothèque, vous pouvez manipuler et animer du texte de différentes manières.

Exemple de texte animé

Les fichiers **TextFly.fla** et **TextFly.as** montrent comment utiliser des chaînes, des formats de texte et des champs texte pour créer une animation. Le fichier d'animation ne contient rien à l'exception de la police. La scène est vide.

La classe **TextFly.as** prend une chaîne et la décompose en caractères, de manière à produire un unique `TextField` et un `Sprite` pour chaque caractère. Elle anime ensuite ces sprites.

La classe commence par définir une série de constantes qui détermineront le comportement de l'animation :

```
package {
    import flash.display.*;
    import flash.text.*;
    import flash.geom.Point;
    import flash.events.*;
    import flash.utils.Timer;

    public class TextFly extends MovieClip {
        // Constantes pour définir l'animation
        static const spacing:Number = 50;
        static const phrase:String = "FlashGameU";
        static const numSteps:int = 50;
        static const stepTime:int = 20;
        static const totalRotation:Number = 360;
        static const startScale:Number = 0.0;
        static const endScale:Number = 2.0;
        static const startLoc:Point = new Point(250,0);
        static const endLoc:Point = new Point(50,100);
        private var letterFormat:TextFormat =
new TextFormat("Monaco",36,0x000000,true,false,
false,null,null,TextFormatAlign.CENTER);
```

Ensuite, elle définit des variables pour contenir les `Sprite` et l'état de l'animation :

```
        // Variables pour tenir registre de l'état de l'animation
        private var letters:Array = new Array();
        private var flySprite:Sprite;
        private var animTimer:Timer;
```

La fonction constructeur crée tous les objets `TextField` et `Sprite`. Elle lance également l'animation en créant un `Timer` :

```
public function TextFly() {
    // Un sprite pour tout contenir
    flySprite = new Sprite();
    addChild(flySprite);

    // Créer toutes les lettres sous forme de champs texte dans des sprites
    for(var i:int=0;i<phrase.length;i++) {
        var letter:TextField = new TextField();
        letter.defaultTextFormat = letterFormat;
        letter.embedFonts = true;
        letter.autoSize = TextFieldAutoSize.CENTER;
        letter.text = phrase.substr(i,1);
        letter.x = -letter.width/2;
        letter.y = -letter.height/2;
        var newSprite:Sprite = new Sprite();
        newSprite.addChild(letter);
        newSprite.x = startLoc.x;
        newSprite.y = startLoc.y;
        flySprite.addChild(newSprite);
        letters.push(newSprite);
    }

    // Lancer l'animation
    animTimer = new Timer(stepTime,numSteps);
    animTimer.addEventListener(TimerEvent.TIMER,animate);
    animTimer.start();
}
Then, with each step of the animation, the rotation and scale of the Sprites will be set:
public function animate(event:TimerEvent) {
    // Où en est l'animation ?
    var percentDone:Number = event.target.currentCount/event.target.repeatCount;

    // Changer position, échelle et rotation
    for(var i:int=0;i<letters.length;i++) {
        letters[i].x = startLoc.x*(1.0-percentDone) + (endLoc.x+spacing*i)*percentDone;
```

```
            letters[i].y = startLoc.y*(1.0-percentDone) + endLoc.y*percentDone;
            var scale:Number = startScale*(1-percentDone)+endScale*percentDone;
            letters[i].scaleX = scale;
            letters[i].scaleY = scale;
            letters[i].rotation = totalRotation*(percentDone-1);
        }
    }
```

La Figure 9.1 présente cette animation en cours d'action.

Il est important de pouvoir contrôler des champs texte et des formats à ce niveau si vous prévoyez de créer des jeux qui utilisent des lettres ou des mots comme pièces de jeu. Nous allons ainsi maintenant examiner un jeu de pendu, sans doute le jeu de lettre le plus simple qu'il soit possible de créer.

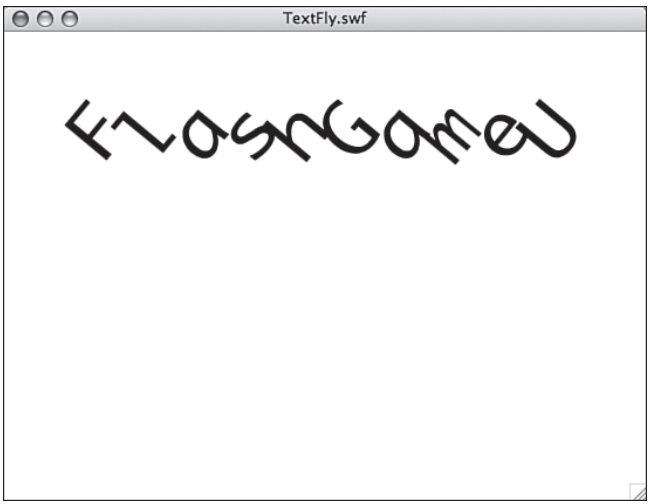


Figure 9.1
Le programme TextFly anime des caractères dans du texte pour les faire voltiger.

Pendu

Codes sources
http://flashgameu.com
A3GPU09_Hangman.zip

Le pendu est un jeu très simple à programmer. Pour ne pas déroger à cette simplicité, nous proposerons dans cet exemple une version dépourvue d'artifices du programme.

Configurer le pendu

En général, le jeu du pendu se joue à deux personnes. La première choisit un mot ou une phrase et dessine chacune des lettres avec un trait de soulignement, puis la seconde s'efforce de deviner les lettres qui le composent.

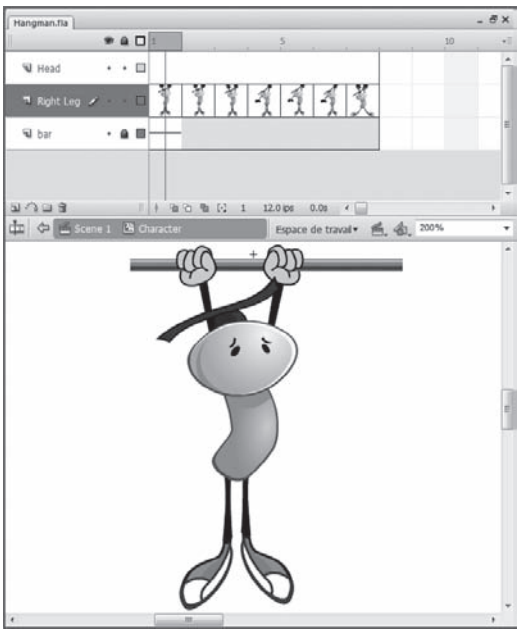
Lorsque le second joueur devine une lettre du mot ou de la phrase, le premier remplit les espaces correspondants où se trouve la lettre. Si le second joueur choisit une lettre qui n'est pas utilisée, le premier dessine une portion supplémentaire d'un pendu dans une image. En général, il suffit de quelques réponses incorrectes pour compléter le pendu, après quoi la partie est perdue.

Dans notre jeu, nous utiliserons une séquence à sept étapes qui diffèrent un petit peu d'un pendu. La Figure 9.2 montre notre caractère de mascotte suspendu à une branche. Après sept choix incorrects, la mascotte tombe.

L'animation **Hangman.fla** contient donc ce seul clip, qui se trouve placé dans la scène vers la droite. Il n'y a sinon rien de spécial avec cette animation, sauf que sa classe est définie comme étant `Hangman`.

Le pendu est apparu au XIXe siècle, vers l'époque où l'on utilisait des potences pour punir les criminels. Cette image inhabituelle est toujours utilisée dans le jeu d'aujourd'hui, même s'il est possible de lui substituer n'importe quel type de séquence en sept étapes.

Figure 9.2
Cette séquence de sept images peut être remplacée par n'importe quelle séquence du même type



La classe *Hangman*

Le jeu tout entier ne fait que cinquante lignes de code. Seules quatre variables de classe sont requises. Il est plaisant de voir qu'un jeu plutôt intéressant peut être créé si rapidement et si facilement en ActionScript 3.0.

Deux chaînes sont requises : l'une pour contenir la phrase et l'autre pour contenir le texte d'affichage, qui commence par des caractères de soulignement à l'endroit où les lettres doivent figurer. Nous aurons ensuite une variable pour contenir une référence au champ texte et une autre pour tenir le registre du nombre d'essais infructueux :

```
package {
    import flash.display.*;
    import flash.text.*;
    import flash.events.*;

    public class Hangman extends Sprite {
        private var textDisplay:TextField;
        private var phrase:String =
"Imagination is more important than knowledge.";
 // - Albert Einstein
        private var shown:String;
        private var numWrong:int;
```

Lorsque la classe démarre, elle crée une copie de la phrase en la faisant traiter par la fonction `replace` avec une expression régulière. L'expression `/[A-Za-z]/g` a n'importe quel caractère de lettre (en somme, de A à Z et de a à z). Elle remplace ces correspondances par un caractère de soulignement :

```
public function Hangman() {
    // Créer une copie du texte avec un _ pour chaque lettre
    shown = phrase.replace(/[A-Za-z]/g,"_");
    numWrong = 0;
```

Le champ texte que nous allons configurer utilisera un format de texte simple pour la police Courier, à 30 points. Il définira la largeur et la hauteur de manière que le texte n'interfère pas avec le graphisme de la mascotte côté droit.

J'ai choisi la police Courier parce qu'il s'agit d'une police à chasse fixe, ce qui signifie que toutes les lettres possèdent la même largeur. Les autres polices possèdent des largeurs variables pour les différentes lettres (par exemple pour le l et le w). En utilisant une police à chasse fixe, nous avons la garantie que les caractères de texte ne changeront pas de position lorsque nous remplaçons les caractères de soulignement par des lettres.

```
        // Configuration du champ texte visible
        textDisplay = new TextField();
        textDisplay.defaultTextFormat = new TextFormat("Courier",30);
        textDisplay.width = 400;
        textDisplay.height = 200;
        textDisplay.wordWrap = true;
        textDisplay.selectable = false;
        textDisplay.text = shown;
        addChild(textDisplay);
```

La fonction `pressKey` sera attribuée à l'événement `KEY_UP` pour la scène :

```
        // Écouter les appuis sur les touches
        stage.addEventListener(KeyboardEvent.KEY_UP,pressKey);
    }
```

Lorsque le joueur enfonce une touche, nous utilisons le code `event.charCode` retourné pour connaître la touche appuyée :

```
    public function pressKey(event:KeyboardEvent) {
        // Déterminer la touche enfoncée
        var charPressed:String = (String.fromCharCode(event.charCode));
```

Une fois que cette lettre est connue, nous recherchons d'éventuelles correspondances dans la phrase. Nous veillons à bien utiliser `toLowerCase` afin que l'appui sur la touche puisse correspondre aux versions en majuscules ou en minuscules de la phrase.

Lorsqu'une correspondance est trouvée, la variable affichée est mise à jour en remplaçant le caractère de soulignement au niveau de cette position par la lettre correspondante de la phrase. La lettre utilisée est ainsi en majuscule ou en minuscule comme elle l'est originellement dans la phrase :

```
        // Boucle pour trouver les lettres correspondantes
        var foundLetter:Boolean = false;
        for(var i:int=0;i<phrase.length;i++) {
            if (phrase.charAt(i).toLowerCase() == charPressed) {
                // Correspondance trouvée, changer la phrase affichée
                shown = shown.substr(0,i)+phrase.substr(i,1)+shown.substr(i+1);
                foundLetter = true;
            }
        }
```

La variable booléenne `foundLetter` est positionnée à `false` lorsque cette recherche démarre et réinitialisée à `true` si une correspondance est trouvée. Si elle reste à `false`, nous savons donc que la lettre ne se trouvait pas dans la phrase et l'image du pendu peut progresser.

Avant cela, nous allons cependant mettre à jour le texte à l'écran en attribuant la valeur `shown` au champ texte :

```
        // Mettre à jour le texte à l'écran
        textDisplay.text = shown;

        // Mettre à jour le pendu
        if (!foundLetter) {
            numWrong++;
            character.gotoAndStop(numWrong+1);
        }
    }
```

Lors des tests dans Flash, assurez-vous de choisir l'option de menu Contrôle > Désactiver les raccourcis clavier. Sans cela, les appuis que vous ferez sur les touches ne parviendront pas à la fenêtre du jeu.

Ce jeu court et simple peut être étendu afin d'inclure les éléments de jeux auxquels nous sommes habitués, comme un écran de départ et un écran de fin de partie. Cet exemple montre qu'il n'est pas nécessaire de passer plus de quelques heures pour créer un jeu amusant.

Considérons maintenant un jeu de mots plus robuste, celui des mots mêlés.

Mots mêlés

Codes sources

http://flashgameu.com

A3GPU09_WordSearch.zip

On pourrait penser que le jeu des mots mêlés est un jeu fort ancien, mais il n'a en réalité fait son apparition qu'au cours des années 1960. Ces jeux sont populaires dans les pages des journaux et dans les revues de vacances.

Les jeux de mots mêlés informatiques peuvent être générés aléatoirement à partir d'une liste de mots ou de dictionnaires. Ce système les rend plus faciles à créer : il vous suffit de fournir une liste de mots.

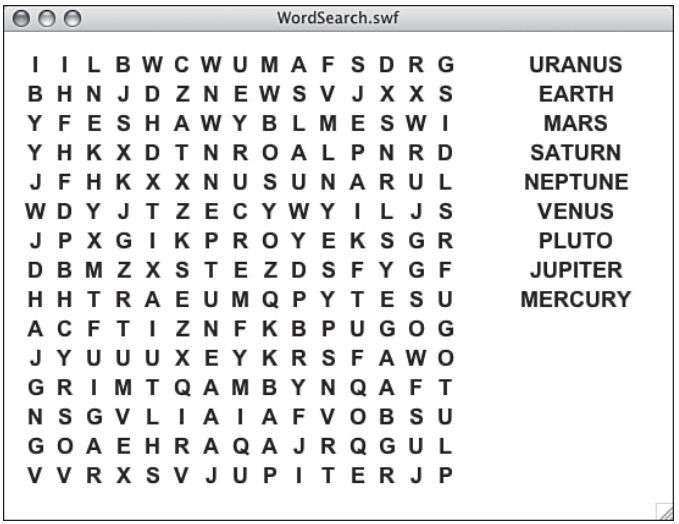
Bien des aspects posent cependant des défis lors de la création d'un jeu de mots mêlés sur ordinateur, par exemple pour l'affichage des lettres, pour la mise en surbrillance horizontale, verticale et diagonale et la conservation d'une liste de mots.

Stratégie de développement

Notre jeu prendra une liste de mots et créera une grille de 15 × 15 lettres représentant ces mots parmi d'autres lettres aléatoires. La Figure 9.3 présente un exemple de grille complète.

Nous allons donc commencer par une grille vide et sélectionner des mots aléatoires dans la liste, des positions aléatoires et des directions aléatoires. Ensuite, nous essaierons d'insérer le mot. S'il ne tient pas ou s'il recouvre des lettres déjà placées dans la grille, le placement est rejeté et une nouvelle tentative est effectuée avec un autre mot, un autre emplacement et une autre direction.

Figure 9.3
La grille au départ, avec la liste de mots sur la droite.



Tous les puzzles de mots mêlés n'utilisent pas les huit directions. Certains n'affichent pas les mots en sens inverse, d'autres n'utilisent pas les diagonales. Tout est affaire de niveau de difficulté. Les puzzles les plus simples conviennent aux jeunes enfants, mais sont bien trop simples pour les adultes.

Cette boucle se répète jusqu'à ce que tous les mots soient placés ou qu'un nombre prédéfini de tentatives ait été réalisé. Cela nous permet d'éviter les cas où il ne reste plus d'espace pour un mot. Il n'est donc pas assuré que tous les mots tiennent dans le puzzle.

Notre exemple n'utilise que neuf mots ; il est donc peu probable que cela se produise, mais les listes plus longues peuvent rencontrer des problèmes. Les listes de mots extrêmement longues n'utiliseront qu'un échantillon des mots à chaque fois, ce qui rend le jeu plus intéressant à jouer plusieurs fois de suite par la même personne.

Une fois que les mots ont été placés, toutes les positions de lettres non utilisées sont remplies avec des lettres aléatoires.

En outre, une liste des mots inclus est placée à droite de l'écran. À mesure que ces mots sont trouvés, ils changent de couleur dans la liste.

Le joueur utilise la souris pour cliquer sur la grille et faire glisser sa sélection. Nous dessinerons une ligne sous les lettres afin d'indiquer celles qui sont sélectionnées. Nous ne le ferons cependant que pour les sélections valides, autrement dit pour les sélections horizontales, verticales ou en diagonale à 45 degrés. La Figure 9.4 montre les différentes directions dans lesquelles un mot peut être disposé.

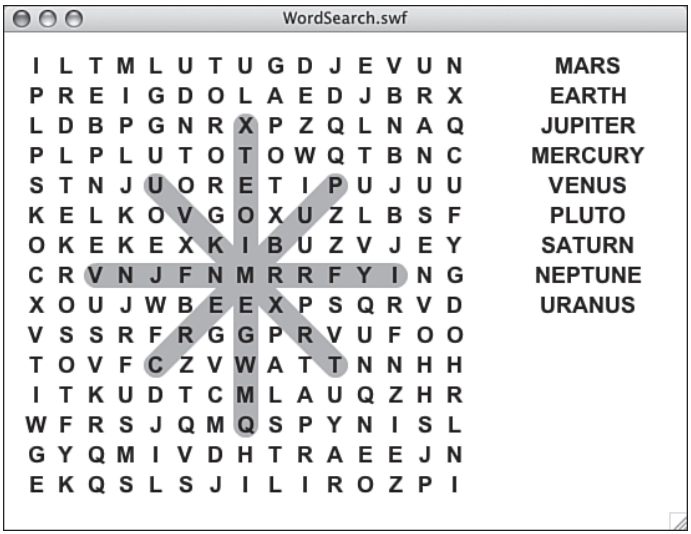


Figure 9.4
Les sélections valides peuvent s'orienter dans huit directions différentes.

Une fois que tous les mots ont été trouvés, la partie se termine.

Définition de la classe

L'image du jeu dans l'animation est complètement vide. Tout sera créé par le code ActionScript. Pour cela, nous aurons besoin des bibliothèques de classes `flash.display`, `flash.text`, `flash.geom` et `flash.events` :

```
package {
    import flash.display.*;
    import flash.text.*;
    import flash.geom.Point;
    import flash.events.*;
```

Plusieurs constantes faciliteront l'ajustement de la taille du puzzle, l'espacement entre les lettres, la taille de la ligne de surlignement, le décalage écran et le format du texte :

```
public class WordSearch extends MovieClip {
    // Constantes
    static const puzzleSize:uint = 15;
    static const spacing:Number = 24;
    static const outlineSize:Number = 20;
    static const offset:Point = new Point(15,15);
    static const letterFormat:TextFormat = new TextFormat("Arial",18,0x000000,true,
    false,false,null,null,TextFormatAlign.CENTER);
```

Pour tenir le registre des mots et de la grille des lettres, nous utiliserons les trois tableaux suivants :

```
    // Mots et grille
    private var wordList:Array;
    private var usedWords:Array;
    private var grid:Array;
```

Le `dragMode` indique si le joueur sélectionne actuellement une séquence de lettres. `startPoint` et `endPoint` définissent cette plage de lettres. `numFound` comptabilise tous les mots trouvés :

```
    // État du jeu
    private var dragMode:String;
    private var startPoint,endPoint:Point;
    private var numFound:int;
```

Ce jeu utilisera plusieurs `Sprite`. Le `gameSprite` contient tout. Les autres contiennent un type d'élément particulier :

```
    // Sprites
    private var gameSprite:Sprite;
    private var outlineSprite:Sprite;
    private var oldOutlineSprite:Sprite;
    private var letterSprites:Sprite;
    private var wordsSprite:Sprite;
```

Créer la grille de recherche des mots

La fonction `startWordSearch` a beaucoup à faire pour créer une grille de puzzle à utiliser dans le jeu. Elle s'appuiera sur la fonction `placeLetters` pour réaliser une partie du travail.

La fonction *startWordSearch*

Pour lancer le jeu, nous allons créer un tableau avec les mots utilisés dans le puzzle. Dans cet exemple, nous utiliserons les neuf planètes, en ignorant les récentes indispositions de l'Union internationale d'astronomie concernant le statut de Pluton :

```
public function startWordSearch() {
    // Liste des mots
    wordList = ("Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus, Neptune,Pluto").split(",");
```

Les `Sprite` sont ensuite créés. Ils le sont dans l'ordre où ils doivent être disposés sur la scène. Les lignes de surbrillance doivent se trouver sous les lettres. Seul le `gameSprite` est ajouté à la scène. Tous les autres sont ajoutés au `gameSprite` :

```
    // Configuration des sprites
    gameSprite = new Sprite();
    addChild(gameSprite);

    oldOutlineSprite = new Sprite();
    gameSprite.addChild(oldOutlineSprite);

    outlineSprite = new Sprite();
    gameSprite.addChild(outlineSprite);

    letterSprites = new Sprite();
    gameSprite.addChild(letterSprites);

    wordsSprite = new Sprite();
    gameSprite.addChild(wordsSprite);
```

Les `Sprite` des lettres seront stockés dans le tableau `grid`. Nous appellerons cependant d'abord `placeLetters` afin d'obtenir un tableau imbriqué avec les caractères à placer dans ces `Sprite`.

En fait, nous nous occupons de décomposer en deux étapes la tâche qui consiste à créer la grille du jeu. La première étape sert à créer une grille virtuelle de lettres sous forme de tableau imbriqué. Elle servira à ajouter les mots de la liste des mots et à remplir le reste avec des lettres aléatoires :

```
    // Tableau de lettres
    var letters:Array = placeLetters();
```

Maintenant que nous savons où les lettres seront placées, nous devons créer les `Sprite`, à raison d'un par lettre. Pour commencer, chaque lettre obtient un `TextField`. Ensuite, ce champ est ajouté à un nouveau `Sprite` :

```
    // Tableau de sprites
    grid = new Array();
    for(var x:int=0;x<puzzleSize;x++) {
```

```
            grid[x] = new Array();
            for(var y:int=0;y<puzzleSize;y++) {

                // Créer un nouveau champ et un sprite pour la lettre
                var newLetter:TextField = new TextField();
                newLetter.defaultTextFormat = letterFormat;
                newLetter.x = x*spacing + offset.x;
                newLetter.y = y*spacing + offset.y;
                newLetter.width = spacing;
                newLetter.height = spacing;
                newLetter.text = letters[x][y];
                newLetter.selectable = false;
                var newLetterSprite:Sprite = new Sprite();
                newLetterSprite.addChild(newLetter);
                letterSprites.addChild(newLetterSprite);
                grid[x][y] = newLetterSprite;
```

En plus d'être créé et ajouté à `letterSprites`, chaque `Sprite` doit se voir associer deux événements : `MOUSE_DOWN` et `MOUSE_OVER`. Le premier démarre une sélection et le second permet à la sélection d'être mise à jour à mesure que le curseur survole des lettres différentes :

```
                // Ajout des écouteurs événementiels
                newLetterSprite.addEventListener(MouseEvent.MOUSE_DOWN, clickLetter);
                newLetterSprite.addEventListener(MouseEvent.MOUSE_OVER, overLetter);
            }
        }
```

Lorsque le joueur relâche le bouton de la souris, nous ne pouvons avoir la garantie qu'il se trouve au-dessus d'une lettre à ce moment. Au lieu d'attacher l'écouteur événementiel `MOUSE_UP` aux lettres, nous l'attachons donc à la scène :

```
        // Écouteur de la scène
        stage.addEventListener(MouseEvent.MOUSE_UP, mouseRelease);
```

La dernière chose à créer est la liste des mots à droite. Il s'agit d'une simple collection d'objets `TextField` placés dans le `Sprite wordsSprite`. Un objet est créé pour chaque mot dans le tableau `usedWords`. Ce tableau sera créé par `placeLetters` et ne contiendra que les mots qui ont pu trouver place dans le puzzle :

```
            // Créer les champs et les sprites de la liste de mots
            for(var i:int=0;i<usedWords.length;i++) {
                var newWord:TextField = new TextField();
```

```
            newWord.defaultTextFormat = letterFormat;
            newWord.x = 400;
            newWord.y = i*spacing+offset.y;
            newWord.width = 140;
            newWord.height = spacing;
            newWord.text = usedWords[i];
            newWord.selectable = false;
            wordsSprite.addChild(newWord);
        }
```

Le jeu est prêt à être joué, hormis qu'il faut encore définir les variables `dragMode` et `numFound` :

```
        // Définition de l'état du jeu
        dragMode = "none";
        numFound = 0;
    }
```

La fonction *placeLetters*

La fonction `placeLetters` réalise plusieurs tâches épineuses. Pour commencer, elle crée une grille vide de 15 × 15 caractères sous forme de tableau imbriqué. Chaque emplacement dans la grille est rempli avec un `*` qui représente un emplacement vide dans le puzzle :

```
    // Placer les mots dans une grille de lettres
    public function placeLetters():Array {

        // Créer une grille vide
        var letters:Array = new Array();
        for(var x:int=0;x<puzzleSize;x++) {
            letters[x] = new Array();
            for(var y:int=0;y<puzzleSize;y++) {
                letters[x][y] = "*";
            }
        }
```

L'étape suivante consiste à créer une copie de `wordList`. Nous souhaitons utiliser une copie et non l'original car nous allons supprimer des mots à mesure que nous les plaçons dans la grille. Nous placerons en outre les mots que nous utilisons dans un nouveau tableau appelé `usedWords` :

```
        // Créer une copie de la liste des mots
        var wordListCopy:Array = wordList.concat();
        usedWords = new Array();
```

Il est maintenant temps d'ajouter des mots dans la grille. Cette opération s'effectue en choisissant un mot aléatoire, un emplacement aléatoire et une direction aléatoire. Ensuite, nous essayons de placer le mot dans la grille, lettre par lettre. Si un conflit se produit (par exemple, le bord de la grille est atteint ou une lettre existante dans la grille ne correspond pas à la lettre que nous souhaitons placer à cet endroit), la tentative est annulée.

Nous poursuivrons nos essais, en insérant parfois un mot, en échouant d'autres fois. Nous continuerons ainsi jusqu'à ce que `wordListCopy` soit vide. Nous comptabiliserons cependant aussi le nombre de tentatives opérées dans `repeatTimes`, qui commencera à 1 000 et décroîtra à chaque tentative. Si `repeatTimes` atteint zéro, nous cessons d'ajouter des mots. À ce point, il est fort probable que tous les mots capables de tenir dans le puzzle s'y trouvent déjà. Nous n'utiliserons pas le reste des mots dans cette configuration aléatoire.

Nous utiliserons la technique qui consiste à étiqueter les boucles afin de pouvoir utiliser la commande `continue` pour forcer le programme à passer directement au début d'une boucle en dehors de la boucle actuelle. Sans ces étiquettes, il serait bien plus difficile de créer le code qui suit.

```
    // Opérer 1 000 tentatives pour ajouter les mots
    var repeatTimes:int = 1000;
    repeatLoop:while (wordListCopy.length > 0) {
        if (repeatTimes-- <= 0) break;

        // Sélectionner un mot, un emplacement et une direction aléatoires
        var wordNum:int = Math.floor(Math.random()*wordListCopy.length);
        var word:String = wordListCopy[wordNum].toUpperCase();
        x = Math.floor(Math.random()*puzzleSize);
        y = Math.floor(Math.random()*puzzleSize);
        var dx:int = Math.floor(Math.random()*3)-1;
        var dy:int = Math.floor(Math.random()*3)-1;
        if ((dx == 0) && (dy == 0)) continue repeatLoop;

        // Vérifier chaque emplacement dans la grille pour voir si le mot tient
        letterLoop:for (var j:int=0;j<word.length;j++) {
            if ((x+dx*j < 0) || (y+dy*j < 0) ||
(x+dx*j >= puzzleSize) || (y+dy*j >= puzzleSize))
 continue repeatLoop;
            var thisLetter:String = letters[x+dx*j][y+dy*j];
            if ((thisLetter != "*") && (thisLetter != word.charAt(j)))
 continue repeatLoop;
```

```
    }

    // Insérer le mot dans la grille
    insertLoop:for (j=0;j<word.length;j++) {
        letters[x+dx*j][y+dy*j] = word.charAt(j);
    }

    // Supprimer le mot de la liste
    wordListCopy.splice(wordNum,1);
    usedWords.push(word);
}
```

Maintenant que nous avons de vrais mots dans la grille, celle-ci doit ressembler à celle de la Figure 9.5, qui représente un jeu sans l'étape suivante.

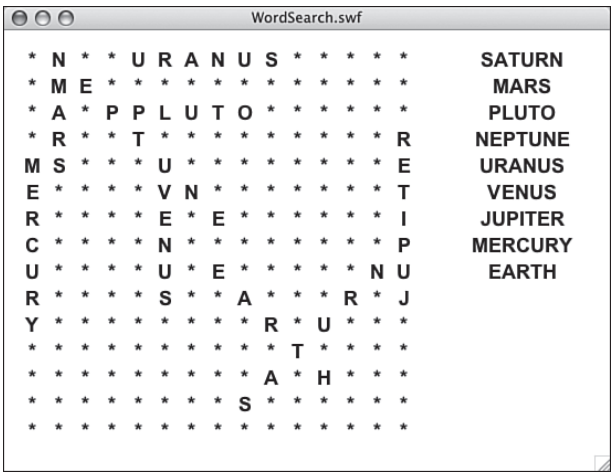


Figure 9.5
*Cette grille contient des caractères * à l'endroit où les lettres aléatoires seront placées.*

Les boucles suivantes examinent chacune des caractères dans la grille et remplacent le * par une lettre aléatoire :

```
// Remplir le reste de la grille avec des lettres aléatoires
for(x=0;x<puzzleSize;x++) {
    for(y=0;y<puzzleSize;y++) {
        if (letters[x][y] == "*") {
            letters[x][y] = String.fromCharCode(65+Math.floor(Math.random()*26));
        }
      }
}
```

Lorsque la fonction `placeLetters` a terminé, elle retourne son tableau afin que les `Sprite` puissent être créés :

```
        return letters;
    }
```

Interaction de l'utilisateur

Nous utiliserons des écouteurs pour surveiller trois actions de souris : le clic, le survol d'un nouveau `Sprite` et le relâchement du bouton de la souris.

Clic de souris

Lorsque le joueur clique sur une lettre, la position sur la grille est déterminée et placée dans `startPoint`. En outre, `dragMode` est positionné à `"drag"`.

La fonction `findGridPoint` retourne un `Point` avec la position de la lettre dans la grille. Nous construirons cette fonction plus tard :

```
// Le joueur clique sur une lettre pour commencer
public function clickLetter(event:MouseEvent) {
    var letter:String = event.currentTarget.getChildAt(0).text;
    startPoint = findGridPoint(event.currentTarget);
    dragMode = "drag";
}
```

Glissement du curseur

À chaque fois que le curseur passe sur une lettre à l'écran, la fonction `overLetter` suivante est appelée. Elle vérifie cependant d'abord que `dragMode` vaut `"drag"`. Le gros de la fonction ne s'exécute donc qu'une fois que le joueur a cliqué sur une lettre.

Le point courant est stocké dans `endPoint`. Maintenant que nous avons un point de départ (`startPoint`) et de fin (`endPoint`), nous pouvons vérifier la plage afin de voir si elle est valide. Nous supposerons qu'elle ne l'est pas, en effaçant le calque graphique `outlineSprite` en premier. S'il s'agit d'une plage valide, `drawOutline` configure cependant le calque graphique `outlineSprite` avec une nouvelle ligne.

En fin de compte, le surlignement est supprimé et redessiné à chaque fois que le curseur change de lettre :

```
// Le joueur sélectionne des lettres
public function overLetter(event:MouseEvent) {
    if (dragMode == "drag") {
        endPoint = findGridPoint(event.currentTarget);

        // Si la plage est valide, afficher la ligne de surbrillance
```

```
            outlineSprite.graphics.clear();
            if (isValidRange(startPoint,endPoint)) {
                drawOutline(outlineSprite,startPoint,endPoint,0xFF0000);
            }
        }
    }
```

Relâchement de la souris

Lorsque le joueur relâche la souris sur une lettre, `dragMode` se voit attribuer la valeur `"none"` et la ligne de surbrillance est effacée. Ensuite, si la plage est valide, deux fonctions sont appelées pour gérer la sélection.

La fonction `getSelectedWord` récupère la plage et retourne les lettres qu'elle contient. Ensuite, la fonction `checkWord` vérifie si ce mot se trouve dans la liste et agit en fonction :

```
    // Bouton de souris relâché
    public function mouseRelease(event:MouseEvent) {
        if (dragMode == "drag") {
            dragMode = "none";
            outlineSprite.graphics.clear();

            // Récupérer le mot et le vérifier
            if (isValidRange(startPoint,endPoint)) {
    var word = getSelectedWord();
                checkWord(word);
            }
        }
    }
```

Fonctions utilitaires

La fonction `findGridPoint` prend un `Sprite` de lettre et détermine l'emplacement auquel il se trouve. Comme les `Sprite` sont créés de toutes pièces, ils ne peuvent pas se voir attribuer des variables dynamiques. Nous ne pouvons donc pas stocker les valeurs `x` et `y` avec chacun des `Sprite`.

Au lieu de cela, nous examinerons donc simplement la grille et trouverons dedans l'élément qui correspond au `Sprite` :

```
    // Lorsque le joueur clique sur une lettre, trouver et retourner l'emplacement x et y
    public function findGridPoint(letterSprite:Object):Point {

        // Parcourir en boucle tous les sprites et trouver celui-ci
        for(var x:int=0;x<puzzleSize;x++) {
```

```
            for(var y:int=0;y<puzzleSize;y++) {
                if (grid[x][y] == letterSprite) {
                    return new Point(x,y);
                }
            }
        }
        return null;
    }
```

Pour déterminer si deux points dans le puzzle constituent une plage valide, nous réalisons trois tests. S'ils se trouvent tous deux sur la même ligne ou la même colonne, la plage est valide. Le troisième test examine la différence `x` et `y`. En cas d'égalité en valeur absolue, la sélection est une diagonale à 45 degrés :

```
// Déterminer si la plage se trouve dans la même ligne, colonne ou dans une diagonale
à 45 degrés
public function isValidRange(p1,p2:Point):Boolean {
    if (p1.x == p2.x) return true;
    if (p1.y == p2.y) return true;
    if (Math.abs(p2.x-p1.x) == Math.abs(p2.y-p1.y)) return true;
    return false;
}
```

Le tracé de la ligne de surbrillance derrière les lettres aurait dû être l'un des problèmes les plus épineux à résoudre de ce jeu. Oui, mais on a parfois de la chance. Grâce aux extrémités arrondies utilisées par défaut pour les traits, il suffit de tracer une ligne d'un emplacement à l'autre, de la rendre suffisamment esthétique et épaisse pour obtenir un trait de surbrillance d'excellente apparence.

Vous remarquerez qu'il est nécessaire de compenser pour placer les extrémités de la ligne au centre des lettres. Les emplacements des lettres correspondent au coin supérieur gauche du `TextField` et donc au coin supérieur gauche du `Sprite` des lettres. La moitié de la constante `spacing` est donc ajoutée afin de compenser ce décalage :

```
// Tracer une ligne épaisse d'un emplacement à un autre
public function drawOutline(s:Sprite,p1,p2:Point,c:Number) {
    var off:Point = new Point(offset.x+spacing/2, offset.y+spacing/2);
    s.graphics.lineStyle(outlineSize,c);
    s.graphics.moveTo(p1.x*spacing+off.x ,p1.y*spacing+off.y);
    s.graphics.lineTo(p2.x*spacing+off.x ,p2.y*spacing+off.y);
}
```

Gérer les mots trouvés

Lorsque le joueur termine une sélection, la première chose à faire est de créer un mot à partir des lettres de la sélection. Pour cela, nous allons déterminer les valeurs `dx` et `dy` entre les deux points, ce qui nous aide à sélectionner les lettres dans la grille.

En commençant à partir de `startPoint`, nous avançons une lettre à la fois. Si la valeur `dx` est positive, chaque étape implique d'avancer d'une colonne vers la droite. Si elle est négative, chaque étape avance d'une colonne vers la gauche. Idem pour `dy` vers le haut et vers le bas. Ce dispositif nous guidera dans les huit directions possibles d'une sélection valide.

Le résultat final correspond à une chaîne de lettres, les mêmes que celles qui apparaissent dans la sélection à l'écran :

```
// Trouver les lettres sélectionnées en fonction des points de départ et de fin
public function getSelectedWord():String {

    // Déterminer dx et dy de la sélection et longueur du mot
    var dx = endPoint.x-startPoint.x;
    var dy = endPoint.y-startPoint.y;
    var wordLength:Number = Math.max(Math.abs(dx),Math.abs(dy))+1;

    // Récupérer chaque caractère de la sélection
    var word:String = "";
    for(var i:int=0;i<wordLength;i++) {
        var x = startPoint.x;
        if (dx < 0) x -= i;
        if (dx > 0) x += i;
        var y = startPoint.y;
        if (dy < 0) y -= i;
        if (dy > 0) y += i;
        word += grid[x][y].getChildAt(0).text;
    }
    return word;
}
```

Une fois que nous connaissons le mot que le joueur pense avoir trouvé, nous pouvons parcourir en boucle le tableau `usedWords` et comparer les lettres trouvées avec les mots de la liste. Nous devons les comparer de gauche à droite et de droite à gauche. Nous ne souhaitons pas imposer au joueur qu'il sélectionne la première lettre en premier, notamment parce que nous allons afficher certains mots en les renversant dans la grille.

Pour inverser un mot, l'un des moyens rapides consiste à utiliser `split` pour convertir la chaîne en un tableau puis `reverse` pour inverser le tableau et `join` pour revenir à une chaîne. `split` et `join` prennent `""` (une chaîne vide) comme séparateur, car nous souhaitons que chaque caractère soit son propre élément dans le tableau :

```
// Comparer le mot à la liste de mots
public function checkWord(word:String) {

    // Parcourir en boucle les mots
    for(var i:int=0;i<usedWords.length;i++) {

        // Comparer le mot
        if (word == usedWords [i].toUpperCase()) {
            foundWord(word);
        }

        // Comparer le mot inversé
        var reverseWord:String = word.split("").reverse().join("");
        if (reverseWord == usedWords [i].toUpperCase()) {
            foundWord(reverseWord);
        }
    }
}
```

Lorsqu'un mot est trouvé, nous souhaitons tracer un trait permanent et supprimer le mot de la liste de droite.

La fonction `drawOutline` peut tracer la ligne sur n'importe quel `Sprite`. Nous lui ferons donc cette fois tracer la ligne sur `oldOutlineSprite` (en utilisant une teinte de rouge plus claire).

Ensuite, nous parcourons en boucle les objets `TextField` dans `wordsSprite` et examinons la propriété `text` de chacun d'entre eux. Si elle correspond au mot, la couleur du `TextField` est remplacée par un gris clair.

Nous augmenterons également `numFound` et appellerons `endGame` si tous les mots ont été trouvés :

```
// Mot trouvé, supprimer de la liste, créer trait de surbrillance permanent
public function foundWord(word:String) {

    // Tracer le trait de surbrillance dans le sprite permanent
    drawOutline(oldOutlineSprite,startPoint,endPoint,0xFF9999);
```

```
        // Trouver le champ texte et le passer en gris
        for(var i:int=0;i<wordsSprite.numChildren;i++) {
            if (TextField(wordsSprite.getChildAt(i)).text.toUpperCase() == word) {
                TextField(wordsSprite.getChildAt(i)).textColor = 0xCCCCCC;
            }
        }

        // Voir si tous les mots ont été trouvés
        numFound++;
        if (numFound == usedWords.length) {
            endGame();
        }
    }
```

La fonction `endGame` conduit simplement le scénario principal à l'image `gameover`. Nous souhaitons non pas encore effacer les sprites du jeu mais plutôt les faire apparaître sous le message "Game Over" et le bouton Play Again.

Pour mieux faire ressortir ces éléments, je les ai placés sur un rectangle uni. Sans cela, ils se fondraient avec la grille de lettres (voir Figure 9.6).

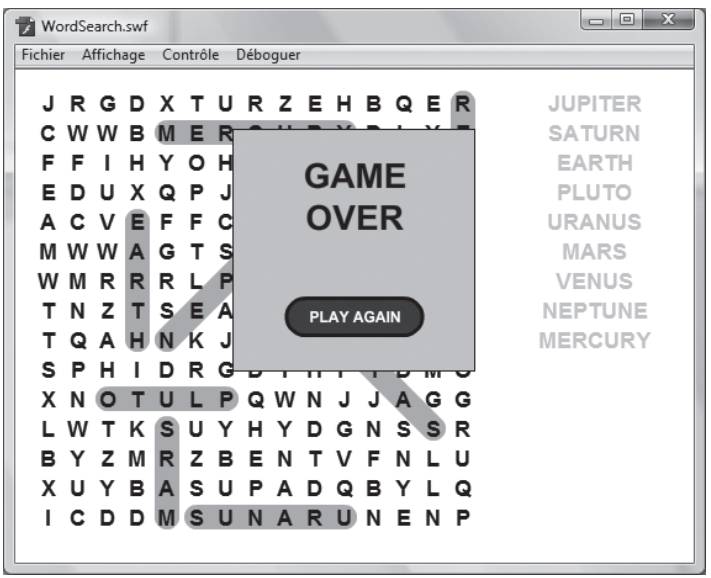


Figure 9.6
Le rectangle permet de mieux faire ressortir le texte Game Over et le bouton.

```
public function endGame() {
    gotoAndStop("gameover");
}
```

Le bouton Play Again appelle `cleanUp` et conduit à l'image `play` pour redémarrer le jeu. Comme nous avons stocké tous nos `Sprite` dans le seul `Sprite` `gameSprite`, nous pouvons nous débarrasser de ce dernier pour effacer la grille :

```
public function cleanUp() {
    removeChild(gameSprite);
    gameSprite = null;
    grid = null;
}
```

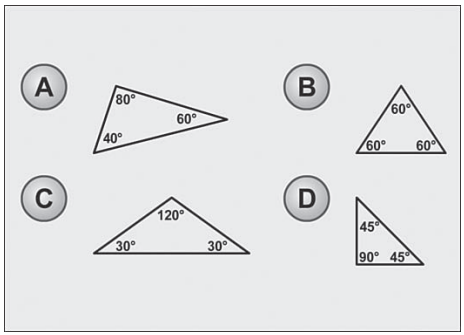
Modifier le jeu

L'intérêt du joueur pour le jeu peut être fortement lié à son intérêt pour les mots. Vous pouvez créer un puzzle sur n'importe quelle thématique. Il suffit simplement d'une liste de mots séparés par des virgules.

En fait, vous pouvez utiliser la technique du Chapitre 2 pour l'inclusion de variables dans le code HTML d'une page Web afin de passer une liste de mots courte. Un unique jeu de mots mêlés pourrait alors être utilisé dans plusieurs pages de votre site avec une liste de mots différente.

Vous pouvez également ajuster aisément les dimensions du puzzle ainsi que la taille et l'espacement des lettres, par exemple afin de proposer des mots mêlés pour les enfants.

L'autre moyen d'obtenir des listes de mots consiste à les importer à partir de fichiers externes. Nous verrons comment importer des données externes au chapitre suivant.



10

Questions et réponses : quiz et jeux de culture générale

Au sommaire de ce chapitre :

- Stocker et récupérer des données de jeu
- Quiz de culture générale
- Quiz version Deluxe
- Quiz en images

Différents jeux peuvent être utilisés à différentes fins, mais il existe peu de jeux qui puissent être utilisés pour des besoins aussi différents que les jeux de quiz. Vous pouvez créer un quiz sur quasiment n'importe quel sujet et à n'importe quel niveau de difficulté. La partie la plus difficile pour la création de jeux de quiz consiste à les rendre intéressants. Après tout, une série de questions à choix multiples, ce n'est rien de plus qu'un test. Or peu de gens sont naturellement enclins à passer des tests.

Les jeux de quiz et de culture générale s'appuient sur des données. Ils se servent de questions et de réponses comme éléments de jeu principaux. Ces données texte sont de préférence stockées dans des fichiers externes et importées de manière dynamique dans le jeu. Nous examinerons les stratégies possibles pour cela avant d'étudier les jeux eux-mêmes.

Après cela, nous créerons un jeu de quiz qui récupère un fichier texte externe et utilise les questions et réponses qu'il contient comme données du jeu. Nous ferons ensuite un pas de plus et utiliserons des images externes dans un jeu de quiz en images.

Stocker et retrouver des données de jeu

Codes sources

http://flashgameu.com

A3GPU10_XMLExamples.zip

Les jeux de culture générale requièrent une liste de questions et de réponses. Le meilleur moyen de faire venir ces données au début d'un jeu consiste à les lire dans un fichier XML.

Flash préfère le XML aux autres types de fichiers de données externes. Flash ne peut en réalité lire que deux types de fichiers : le XML et une liste d'attributions de variables. Le second format n'est véritablement utile que pour les petites tâches. Le XML peut être utilisé pour de très grandes bases de données s'il le faut.

Comprendre les données XML

XML est l'acronyme d'eXtensible Markup Language (langage de balisage extensible). Son rôle consiste à proposer un format simple pour l'échange d'informations entre systèmes.

Si vous n'avez jamais vu de fichier XML auparavant mais que vous ayez travaillé en HTML, vous remarquerez une similarité. Les signes inférieur à et supérieur à sont utilisés en XML pour entourer des mots-clés de définition appelés balises.

Observez l'exemple suivant :

```
<trivia>
    <item category="Entertainment">
        <question>Who is known as the original drummer of the Beatles?</question>
        <answers>
            <answer>Pete Best</answer>
            <answer>Ringo Starr</answer>
            <answer>Stu Sutcliffe</answer>
            <answer>George Harrison</answer>
        </answers>
        <hint>Was fired before the Beatles hit it big.</hint>
        <fact>Pete stayed until shortly after their first audition for EMI in 1962, but
        was fired on August 16th of that year, to be replaced by Ringo Starr.</fact>
    </item>
</trivia>
```

Ce fichier XML représente un quiz à une question. Les données suivent un format imbriqué : des balises sont placées à l'intérieur d'autres balises. Par exemple, le document entier correspond à un objet `<trivia>`. À l'intérieur de cet objet figure un `<item>`. Dans cet objet `<item>` figure une `<question>`, un objet `<answers>` avec quatre objets `<answer>`, un objet `<hint>` et un objet `<fact>`.

Les objets individuels des documents XML sont aussi appelés des nœuds. Un nœud peut contenir simplement des données ou avoir plusieurs nœuds enfants. Certains nœuds se voient associer des données supplémentaires, comme le nœud `item`, qui dans cet exemple possède une `category`. On les appelle des attributs.

Vous pouvez placer un document XML directement à l'intérieur de votre code ActionScript 3.0. Par exemple, l'animation d'exemple **xmlExample.fla** en contient un dans le script de l'image 1 :

```
var myXML:XML =
    <trivia>
        <item category="Entertainment">
            <question>Who is known as the original drummer of the Beatles?</question>
            <answers>
                <answer>Pete Best</answer>
                <answer>Ringo Starr</answer>
                <answer>Stu Sutcliffe</answer>
                <answer>George Harrison</answer>
```

```
            </answers>
            <hint>Was fired before the Beatles hit it big.</hint>
            <fact>Pete stayed until shortly after their first audition for EMI in 1962, but
            was fired on August 16th of that year, to be replaced by Ringo Starr.</fact>
        </item>
    </trivia>
```

Vous remarquerez qu'aucun guillemet ni parenthèse n'est requis autour des données XML. Elles peuvent tout simplement figurer à l'intérieur du code ActionScript 3.0 (vous imaginez cependant combien cela peut devenir incommode si les données deviennent plus longues).

Maintenant que nous avons des données XML dans un objet XML, nous pouvons nous amuser à en extraire des informations.

La gestion des données XML a été considérablement améliorée avec ActionScript 3.0. Auparavant, il fallait utiliser des instructions plus complexes pour trouver un nœud spécifique dans les données. Le nouvel objet `XML` dans ActionScript 3.0 diffère de l'objet `XML` sous ActionScript 2.0, ce qui signifie que vous ne pouvez pas effectuer la conversion directe de l'un à l'autre. Faites donc attention aux anciens exemples de code qui pourraient toujours se trouver au format ActionScript 2.0.

Pour obtenir le nœud `question` dans les données, vous procéderiez ainsi :

```
trace(myXML.item.question);
```

Voilà qui est plutôt simple. Pour obtenir un attribut, il suffit d'utiliser la fonction `attribute` :

```
trace(myXML.item.attribute("category"));
```

Le symbole `@` peut être utilisé comme raccourci pour obtenir l'attribut. Au lieu de `myXML.item.attribute("category")`, vous pouvez donc aussi écrire `myXML.item.@category`.

Dans le cas du nœud `<answers>`, il y a quatre réponses (`answer`). Celles-ci peuvent être traitées comme un tableau en y accédant avec des crochets :

```
trace(myXML.item.answers.answer[1]);
```

L'obtention du nombre de nœuds à l'intérieur d'un autre nœud, par exemple du nombre de nœuds `<answer>`, est un peu moins évidente. Vous devez procéder de la manière suivante :

```
trace(myXML.item.answers.child("*").length());
```

La fonction `child` retourne un enfant d'un nœud spécifié par une chaîne ou un nombre. En revanche, `"*"` retourne tous les nœuds enfants. `length()` fournit ensuite le nombre de nœuds enfants. Si vous

essayez simplement d'obtenir la longueur d'un nœud avec `length()`, vous obtiendrez le résultat 1, car un nœud fait toujours un de longueur.

Maintenant que vous savez comment trouver votre chemin dans les données XML, commençons à gérer des documents XML plus grands importés depuis des fichiers externes.

Importer des fichiers XML externes

Lorsque le XML est enregistré sous forme de fichier, il est semblable à un fichier de texte brut. Vous pouvez d'ailleur ouvrir vos fichiers XML avec pratiquement n'importe quel éditeur de texte. Le fichier **trivia1.xml** est un fichier court contenant simplement dix éléments de quiz.

Pour ouvrir et lire un fichier externe, nous allons utiliser les objets `URLRequest` et `URLLoader`. Nous définirons ensuite un événement à déclencher lorsque le fichier aura été chargé.

L'exemple de code suivant présente un fragment de code de chargement XML provenant de **xmlimport.as**. La fonction constructeur crée un objet `URLRequest` avec le nom du fichier XML. Ensuite, `URLLoader` lance le téléchargement.

Vous pouvez passer n'importe quelle URL valide à `URLRequest`. L'usage d'un simple nom de fichier comme nous l'avons fait ici implique que le fichier se trouve aux côtés de l'animation Flash SWF, dans le même dossier. Vous pouvez cependant séparer un sous-dossier ou même utiliser . . / et d'autres fonctionnalités de cheminement pour indiquer une URL relative. Vous pouvez enfin aussi spécifier des URL absolues, qui fonctionnent aussi bien sur le serveur et lors du test local sur votre ordinateur.

Nous allons attacher un écouteur au `URLLoader`. Cet écouteur appellera `xmlLoaded` lorsque le fichier aura été complètement téléchargé :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.net.URLLoader;
    import flash.net.URLRequest;

    public class xmlimport extends MovieClip {
        private var xmldata:XML;

        public function xmlimport() {
            xmldata = new XML();
            var xmlURL:URLRequest = new URLRequest("xmltestdata.xml");
            var xmlLoader:URLLoader = new URLLoader(xmlURL);
            xmlLoader.addEventListener(Event.COMPLETE,xmlLoaded);
        }
```

La fonction `xmlLoaded` prend les données chargées de `event.target.data` et les convertit en XML pour les stocker dans `xmldata`. En guise de test, plaçons la seconde réponse à la première question dans la fenêtre Sortie :

```
            function xmlLoaded(event:Event) {
                xmldata  = XML(event.target.data);
                trace(xmldata.item.answers.answer[1]);
                trace("Data loaded.");
            }
        }
    }
```

Les objets XML, comme les tableaux, sont indicés à zéro. La première réponse du précédent exemple se trouve donc à la position 0 et la seconde, à la position 1.

Capturer les erreurs de chargement

Puisqu'on n'est jamais à l'abri d'une erreur, il est toujours utile de mettre en place un mécanisme de vérification des erreurs. Vous pouvez le faire en ajoutant un autre événement à `URLLoader` :

```
xmlLoader.addEventListener(IOErrorEvent.IO_ERROR,xmlLoadError);
```

Ensuite, vous pouvez obtenir le message d'erreur de l'événement retourné à `xmlLoadError` :

```
function xmlLoadError(event:IOErrorEvent) {
    trace(event.text);
}
```

Je n'indiquerai cependant pas à l'utilisateur final le message d'erreur verbatim. Par exemple, si vous supprimez simplement le fichier et essayez de lancer l'animation, vous obtenez l'erreur suivante suivie par le nom de fichier :

```
Error #2032: Stream Error. URL: file:
```

Ce n'est pas le genre de message d'erreur que vous souhaiterez présenter au joueur. Il vaudrait mieux quelque chose du genre "Unable to load game file" (le fichier du jeu n'a pas pu être chargé).

Vous savez maintenant comment récupérer des documents XML plus imposants, comme ceux dont vous aurez besoin pour créer des jeux de culture générale.

Quiz de culture générale

Codes sources

http://flashgameu.com

A3GPU10_TriviaGame.zip

Le quiz de culture générale aussi baptisé Trivia est devenu pour la première fois une forme de divertissement dans les années 1950, avec l'apparition de la télévision. Les shows télévisés de quiz sont devenus de plus en plus populaires au fil des ans.

Dans les années 1980, les jeux de société comme le Trivial Pursuit ont rencontré un franc succès et permis aux gens de jouer à des jeux de culture générale en plus de les regarder. Ces jeux sont rapidement devenus disponibles sur les ordinateurs et sur Internet.

Les jeux de culture générale sont un bon moyen de décliner des jeux sur tous les sujets. Votre site Web concerne les pirates ? Créez un jeu de quiz sur les pirates. Vous créez un CD-ROM pour une conférence à Francfort ? Ajoutez un jeu de quiz avec des faits intéressants concernant la ville.

Créons d'abord un jeu de quiz simple, puis ajoutons des gadgets au fur et à mesure.

Concevoir un jeu de quiz simple

Le jeu de quiz élémentaire correspond simplement à une série de questions. Le joueur lit une question, puis sélectionne une réponse parmi plusieurs possibles. Les joueurs ont un point ou une forme de récompense en cas de bonne réponse. Ensuite, le jeu passe à la question suivante.

Nous construirons ce jeu comme tous les autres : avec trois images, l'action prenant place dans l'image du milieu. Dans le cas présent, l'action désigne une série d'éléments de texte et de boutons. Nous commencerons par demander au joueur s'il est prêt à commencer. Il cliquera sur un bouton pour démarrer (voir Figure 10.1).

Le joueur se voit ensuite présenter une question et quatre réponses. Il doit choisir une réponse. Si la réponse qu'il choisit est juste, le message "You Got It!" (trouvé!) apparaît. En cas d'erreur, il obtient le message "Incorrect".

Dans un cas comme dans l'autre, le joueur voit apparaître un autre bouton sur lequel il doit cliquer pour passer à la question suivante.

Ouvrez **TriviaGame.fla** et essayez de jouer afin de mieux comprendre le fonctionnement du jeu. À présent, construisons le jeu.

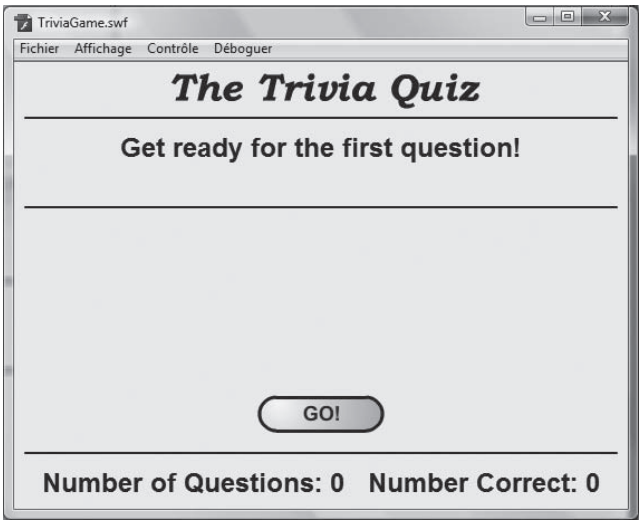


Figure 10.1
Au début du jeu, le joueur voit apparaître un bouton sur lequel il doit cliquer avant d'obtenir la première question.

Configurer l'animation

Le fichier d'animation n'utilise que deux images au lieu des trois que nous avons eu l'habitude d'utiliser. Nous aurons besoin d'un nouvel élément dans la bibliothèque de notre animation pour créer le jeu de quiz. Il s'agira d'un cercle contenant une lettre et qui s'affichera à côté d'une réponse. La Figure 10.2 présente ce clip.

Le champ texte dans le clip `Circle` est appelé `letter`. Nous en créerons quatre comme celui-là, un pour chaque réponse, et les placerons à côté du texte de la réponse. La lettre sera chaque fois différente : A, B, C ou D.

Si vous observez attentivement la Figure 10.2, vous remarquerez que le point d'alignement du clip est décalé vers le haut et la droite. Ce point correspond à l'emplacement 0,0 du champ texte qui ira à côté. Nous pouvons ainsi attribuer à `Circle` *et au champ texte de réponse le même emplacement et les faire apparaître côte à côte au lieu de l'un sur l'autre.*

La même technique de l'image d'arrière-plan et du champ texte sera utilisée dans le clip `GameButton`. Cela nous permettra d'utiliser le même clip de bouton pour différents boutons dans le jeu.

L'animation contient également des graphismes d'arrière-plan, dont un titre et des lignes horizontales (voir Figure 10.1).

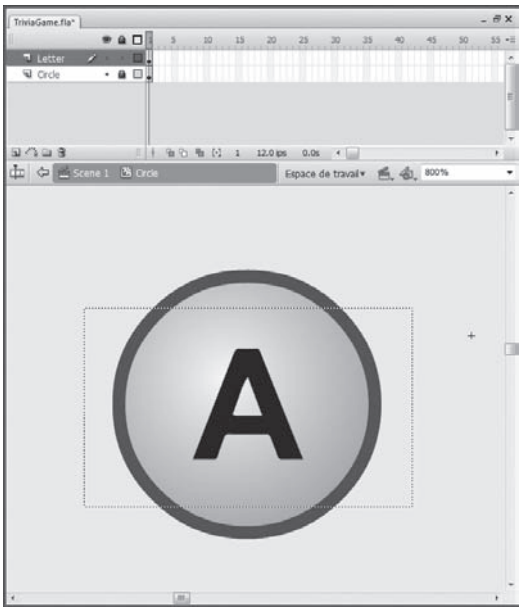


Figure 10.2
Le clip Circle contient un champ texte dynamique et un cercle d'arrière-plan.

Configurer la classe

Comme ce jeu charge les données du quiz à partir d'un fichier externe, nous avons besoin de certaines parties de la bibliothèque `flash.net` pour utiliser les fonctions `URLLoader` et `URLRequest` :

```
package {
    import flash.display.*;
    import flash.text.*;
    import flash.events.*;
    import flash.net.URLLoader;
    import flash.net.URLRequest;
```

Le jeu utilisera un certain nombre de variables. Nous placerons les données chargées depuis le fichier dans `dataXML`. Nous avons également différents formats texte et des références à des champs texte dynamiques que nous allons créer :

```
    public class TriviaGame extends MovieClip {

        // Données de question
        private var dataXML:XML;

        // Formats texte
        private var questionFormat:TextFormat;
```

```
        private var answerFormat:TextFormat;
        private var scoreFormat:TextFormat;

        // Champs texte
        private var messageField:TextField;
        private var questionField:TextField;
        private var scoreField:TextField;
```

L'idée pour les sprites est d'avoir un `gameSprite` qui contient tout. À l'intérieur, nous aurons un `questionSprite` contenant tous les éléments d'une unique question de quiz : un champ texte pour la question et d'autres sprites pour les réponses. Les `answerSprites` contiendront les champs texte et les clips `Circle` pour chaque réponse, qui seront stockés dans leur propre sprite. Nous n'avons cependant pas besoin d'une variable de classe pour les référencer, car ils seront proprement stockés dans le sprite `answerSprites`.

Nous aurons aussi une référence pour le bouton `GameButton`, afin que lorsque nous créons un bouton nous puissions utiliser cette référence pour le supprimer :

```
        // Sprites et objets
        private var gameSprite:Sprite;
        private var questionSprite:Sprite;
        private var answerSprites:Sprite;
        private var gameButton:GameButton;
```

Pour tenir le registre de l'état du jeu, nous avons besoin de `questionNum`, qui mémorise la question à laquelle nous nous trouvons, de `numCorrect`, qui consigne le score du joueur, et de `numQuestionsAsked`, qui concerne également la détermination du score du joueur.

Pour mémoriser la question posée, nous placerons les quatre réponses dans un ordre quelconque dans le tableau `answers`. Avant de les mélanger, nous tiendrons cependant note de la première réponse d'origine, que nous savons correspondre à la bonne réponse, dans la variable `correctAnswer` :

```
        // Variables d'état du jeu
        private var questionNum:int;
        private var correctAnswer:String;
        private var numQuestionsAsked:int;
        private var numCorrect:int;
        private var answers:Array;
```

La fonction constructeur créera le `gameSprite` puis configurera les trois objets `TextFormat` :

```
    public function startTriviaGame() {

        // Créer le sprite du jeu
        gameSprite = new Sprite();
        addChild(gameSprite);
```

```
// Définition des formats texte
questionFormat = new TextFormat("Arial",24,0x330000, true,false,false,null,null,"center");
answerFormat = new TextFormat("Arial",18,0x330000, true,false,false,null,null,"left");
scoreFormat = new TextFormat("Arial",18,0x330000, true,false,false,null,null,"center");
```

Il n'existe pas de moyen de dupliquer un objet `TextFormat`. *Si vous tapez simplement* `answerFormat = questionFormat` *puis apportez une modification à l'un des deux formats, l'autre change également. Il est donc important de créer un nouvel objet* `TextFormat` *pour chacune des variables.*

Vous pouvez en revanche définir une variable temporaire comme `myFont` *en lui attribuant une valeur telle que "Arial" puis utiliser myFont à la place de "Arial" dans chacune des déclarations* `TextFormat`. *Vous pourrez alors modifier la police utilisée dans le jeu en opérant un unique changement à un seul emplacement du code.*

Lorsque le jeu démarre, les champs texte `scoreField` et `messageField` sont créés. Au lieu de créer un `TextField`, de l'ajouter avec `addChild` et de positionner chacune de ses propriétés pour chacun des éléments de texte requis, nous créons une fonction utilitaire appelée `createText` qui se charge de cette tâche pour nous en une ligne de code. Par exemple, le `messageField` contiendra le texte `"Loading Questions …"` en utilisant le format `questionFormat`. Il le place dans le `gameSprite` à 0,50 avec une largeur de 550. Nous examinerons `createText` plus tard :

```
    // Créer les champs texte du score et du message de départ
    scoreField = createText("",questionFormat,gameSprite,0,360,550);
    messageField = createText("Loading Questions...",questionFormat,gameSprite,0,50,550
);
```

Une fois que l'état du jeu est défini, nous appelons `showScore` pour placer le texte du score en bas de l'écran. Nous étudierons cela plus tard également.

Nous appelons `xmlImport` pour récupérer les données du quiz :

```
    // Configuration de l'état et chargement des questions
    questionNum = 0;
    numQuestionsAsked = 0;
    numCorrect = 0;
    showScore();
    xmlImport();
}
```

Le texte `"Loading Questions …"` apparaîtra à l'écran et y restera jusqu'à ce que le document XML ait été lu. Lors du test de l'animation, cette étape peut prendre moins de 1 seconde. Une fois que le jeu se trouvera sur le serveur, elle devrait être un peu plus longue, selon la réactivité de la connexion du joueur.

Charger les données du quiz

Les questions sont chargées en utilisant des fonctions analogues à l'exemple du début de ce chapitre. Pour ne pas encombrer le code de cet exemple, aucune vérification d'erreur n'est opérée. Le fichier **trivia1.xml** contient dix éléments :

```
// Lancer le chargement des questions
public function xmlImport() {
    var xmlURL:URLRequest = new URLRequest("trivia1.xml");
    var xmlLoader:URLLoader = new URLLoader(xmlURL);
    xmlLoader.addEventListener(Event.COMPLETE, xmlLoaded);
}
```

Une fois le chargement terminé, les données sont placées dans `dataXML`. Ensuite, le message texte qui affichait `"Loading Questions …"` est supprimé. Il est remplacé par un nouveau message indiquant "Get ready for the first question!" (préparez-vous pour la première question !).

Une autre fonction utilitaire est appelée pour créer un `GameButton`. Une étiquette de bouton `"GO!"` est placée à l'intérieur du bouton. Nous examinerons `showGameButton` un peu plus loin dans ce chapitre :

```
// Questions chargées
public function xmlLoaded(event:Event) {
    dataXML  = XML(event.target.data);
    gameSprite.removeChild(messageField);
    messageField = createText("Get ready for the first question!",questionFormat,
                              gameSprite,0,60,550);
    showGameButton("GO!");
}
```

Le jeu attend maintenant que le joueur clique sur le bouton.

Texte du message et bouton de jeu

Plusieurs fonctions utilitaires sont requises dans ce jeu pour créer des champs texte et des boutons. Elles permettent de réduire sensiblement la longueur du code. Nous n'aurons pas à répéter les mêmes réglages `TextField`, `addChild`, `x` et `y` à chaque fois que nous créons un champ texte.

`createText` s'occupe de récupérer une série de paramètres et de créer un nouvel objet `TextField`. Elle positionne les valeurs `x`, `y`, `width` et `TextFormat` en leur attribuant les valeurs passées en paramètre. Elle définit également des paramètres de constante comme `multiline` et `wordWrap` qui seront les mêmes pour tout ce que nous créons dans le jeu.

L'alignement du texte dans le champ variera : il sera parfois centré et d'autres fois justifié à gauche. Ce réglage est inclus dans `TextFormat`. Nous souhaitons cependant attribuer la valeur qui convient à la propriété `autoSize` du champ ; nous réalisons donc un test et attribuons à `autoSize` la valeur `TextFieldAutoSize.LEFT` ou la valeur `TextFieldAutoSize.RIGHT` en fonction.

Pour finir, nous définissons le texte du champ et ajoutons ce dernier au sprite passé dans un autre paramètre. Le `TextField` est retourné par la fonction ; nous pouvons donc définir une variable pour le référencer dont nous nous servirons pour le supprimer par la suite :

```
// Créer un champ texte
public function createText(text:String, tf:TextFormat,
s:Sprite, x,y: Number, width:Number): TextField {
    var tField:TextField = new TextField();
    tField.x = x;
    tField.y = y;
    tField.width = width;
    tField.defaultTextFormat = tf;
    tField.selectable = false;
    tField.multiline = true;
    tField.wordWrap = true;
    if (tf.align == "left") {
        tField.autoSize = TextFieldAutoSize.LEFT;
    } else {
        tField.autoSize = TextFieldAutoSize.CENTER;
    }
    tField.text = text;
    s.addChild(tField);
    return tField;
}
```

Contrairement aux autres champs texte, le champ `scoreField` ne sera pas créé, supprimé puis recréé durant le jeu. Il sera simplement créé une fois et placé en bas de l'écran. Ensuite, nous utiliserons `showScore` pour mettre à jour le texte dans le champ :

```
// Met à jour le score
public function showScore() {
    scoreField.text = "Number of Questions: "+numQuestionsAsked+"   Number Correct:
    "+numCorrect;
}
```

De la même manière que `createText` nous permet de créer différents types de champs texte avec une fonction, `showGameButton` nous permet de créer différents boutons. Elle prend `buttonLabel` en paramètre et s'en sert pour définir le texte de l'étiquette à l'intérieur du bouton.

La variable `gameButton` est déjà une propriété de classe. Elle sera donc disponible par la suite pour `removeChild`. Nous ajouterons un écouteur événementiel à ce bouton afin qu'il appelle `pressGameButton` lorsque le joueur clique dessus. Cette action servira à passer à l'étape suivante du jeu :

```
// Demander au joueur s'il est prêt pour la question suivante
public function showGameButton(buttonLabel:String) {
    gameButton = new GameButton();
```

```
        gameButton.label.text = buttonLabel;
        gameButton.x = 220;
        gameButton.y = 300;
        gameSprite.addChild(gameButton);
        gameButton.addEventListener(MouseEvent.CLICK,pressedGameButton);
    }
```

Avec la programmation de haut en bas, il est nécessaire de tester chaque fragment de code à mesure que vous l'écrivez. Malheureusement, l'exemple de code précédent génère une erreur parce que `pressedGameButton` n'existe pas encore. À ce stade, il m'arrive généralement de créer une fonction `pressedGameButton` factice qui ne contient pas de code. Cela me permet de tester d'abord l'emplacement du bouton, avant d'avoir besoin de programmer la procédure qui survient lorsque le joueur clique sur le bouton.

Passer à l'étape suivante

Lorsque le joueur clique sur un bouton, le jeu doit passer à l'étape suivante. La plupart du temps, cela implique de présenter une nouvelle question, mais lorsqu'il n'y a plus de question la partie se termine.

Pour commencer, nous devons supprimer la question précédente. S'il s'agit de la première question, `questionSprite` n'aura pas encore été créé. Nous devons donc vérifier l'existence de `questionSprite` et ne le supprimer que s'il se trouve bien là :

```
    // Le joueur est prêt
    public function pressedGameButton(event:MouseEvent) {
        // Supprimer la question
        if (questionSprite != null) {
            gameSprite.removeChild(questionSprite);
        }
```

D'autres choses doivent être supprimées également. Le message et le bouton qui restent suite à la pause avant ou entre les questions sont supprimés :

```
        // Supprimer le bouton et le message
        gameSprite.removeChild(gameButton);
        gameSprite.removeChild(messageField);
```

À présent, nous devons déterminer si toutes les questions ont été posées. Si c'est le cas, nous passons directement à l'image `gameover`. L'écran est déjà vide car la question précédente, le message et le bouton ont été supprimés.

Si les questions ne sont pas terminées, nous appelons `askQuestion` pour afficher la question suivante :

```
        // Poser la question suivante
        if (questionNum >= dataXML.child("*").length()) {
            gotoAndStop("gameover");
        } else {
            askQuestion();
        }
    }
```

Afficher les questions et les réponses

La fonction `askQuestion` récupère la question suivante dans les données du quiz et l'affiche. Elle place tout ce qu'elle crée dans le sprite `questionSprite` afin d'en faciliter la suppression par la suite. La Figure 10.3 présente l'écran après qu'une question a été affichée.

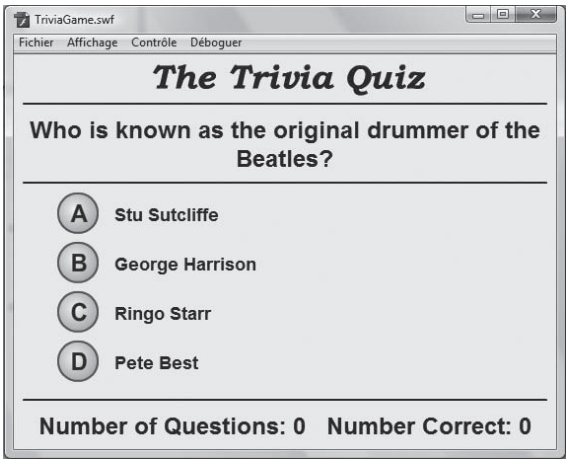


Figure 10.3
La question et les quatre réponses sont affichées dans le sprite questionSprite, qui couvre l'essentiel du milieu de l'écran.

```
// Configurer la question
public function askQuestion() {
    // Préparer nouveau sprite de question
    questionSprite = new Sprite();
    gameSprite.addChild(questionSprite);
```

La question elle-même apparaîtra dans un champ isolé vers le haut de l'écran :

```
        // Créer le champ texte pour la question
        var question:String = dataXML.i tem[questionNum].question;
        questionField = createText(question,questionFormat,questionSprite,0,60,550);
```

Avant de placer les réponses, nous devons les mélanger. La première parmi les données d'origine correspond à la bonne réponse, aussi en stockons-nous une copie dans `correctAnswer`. Ensuite, nous appelons `shuffleAnswers` pour obtenir un tableau de toutes les réponses dans un ordre aléatoire :

```
// Créer le sprite pour les réponses, obtenir la bonne réponse et mélanger
correctAnswer = dataXML.item[questionNum].answers.answer[0];
answers = shuffleAnswers(dataXML.item[questionNum].answers);
```

Les réponses se trouvent dans un sous-sprite de `questionSprite` appelé `answerSprites`. Un objet `TextField` et un objet `Circle` sont créés pour chaque réponse. Les objets `Circle` se voient tous attribuer des lettres différentes, de A à D. Les deux sont placés au même endroit, mais le clip `Circle` a été conçu pour apparaître à gauche de son point d'alignement, tandis que le texte apparaîtra à droite.

Le texte et `Circle` seront tous deux livrés dans un nouveau sprite individuel, lequel sprite se verra attribuer un écouteur `CLICK` afin de pouvoir réagir comme un bouton :

```
// Placer chaque réponse dans un nouveau sprite avec une icône de cercle
answerSprites = new Sprite();
for(var i:int=0;i<answers.length;i++) {
    var answer:String = answers[i];
    var answerSprite:Sprite = new Sprite();
    var letter:String = String.fromCharCode(65+i); // A-D
    var answerField:TextField =  createText(answer,answerFormat,answerSprite,0,0,450);
    var circle:Circle = new Circle(); // from Library
    circle.letter.text = letter;
    answerSprite.x = 100;
    answerSprite.y = 150+i*50;
    answerSprite.addChild(circle);
    answerSprite.addEventListener(MouseEvent.CLICK,clickAnswer);
    answerSprite.buttonMode = true;
    answerSprites.addChild(answerSprite);
}
questionSprite.addChild(answerSprites);
}
```

Pour convertir un nombre en une lettre, nous utilisons `String.fromCharCode(65+i)`. *Le caractère 65 produit A, le caractère 66 produit B, etc.*

La fonction `shuffleAnswers` récupère en entrée une `XMLList` correspondant au type de données retourné en demandant `dataXML.item[questionNum].answers`. Elle parcourt les données en boucle en supprimant à chaque fois un élément au hasard de la liste et en le plaçant dans un tableau. Elle retourne ensuite ce tableau de réponses trié de manière aléatoire :

```
// Récupérer toutes les réponses et les mélanger dans un tableau
public function shuffleAnswers(answers:XMLList) {
    var shuffledAnswers:Array = new Array();
    while (answers.child("*").length() > 0) {
        var r:int = Math.floor(Math.random()*answers.child("*").length());
        shuffledAnswers.push(answers.answer[r]);
        delete answers.answer[r];
    }
    return shuffledAnswers;
}
```

Juger les réponses

Toutes les fonctions considérées jusqu'ici s'occupaient de configurer le jeu. À présent, le joueur se voit enfin présenter la question (voir Figure 10.3).

Lorsque le joueur clique sur l'une des quatre réponses, nous appelons `clickAnswer`. Cette fonction s'occupe en premier lieu de récupérer le texte de la réponse sélectionnée. L'objet `TextField` est le premier enfant de `currentTarget`. Nous récupérons donc la valeur de sa propriété `text` et la plaçons dans `selectedAnswer`.

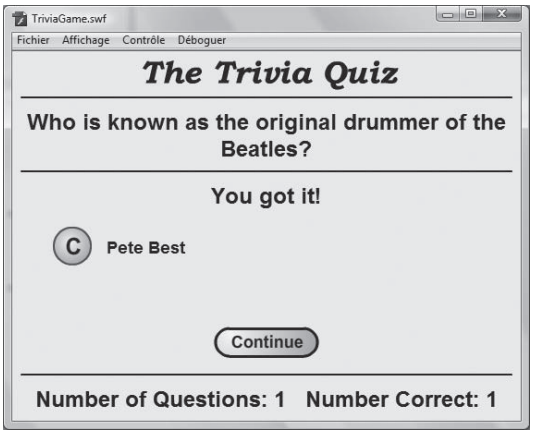
Cette valeur est ensuite comparée avec celle de `correctAnswer` que nous avons stockée lorsque la question s'est affichée. Si le joueur a choisi la bonne réponse, nous créditons `numCorrect` d'une unité. Un nouveau message texte est affiché dans les deux cas :

```
// Le joueur sélectionne une réponse
public function clickAnswer(event:MouseEvent) {

    // Récupérer le texte de la réponse sélectionnée et comparer
    var selectedAnswer = event.currentTarget.getChildAt(0).text;
    if (selectedAnswer == correctAnswer) {
        numCorrect++;
        messageField = createText("You got it!",questionFormat,gameSprite,0,140,550);
    } else {
        messageField = createText("Incorrect! The correct answer was:", questionFormat,
        gameSprite,0,140,550);
    }
    finishQuestion();
}
```

Toutes les réponses sont ensuite examinées. La fonction `finishQuestion` parcourt en boucle chaque sprite. La réponse correcte est déplacée à une position `y` qui la situe au milieu. Tous les écouteurs événementiels sont supprimés également. Les autres réponses sont rendues invisibles. La Figure 10.4 montre l'écran à ce stade.

Figure 10.4
La bonne réponse est déplacée au milieu et accompagnée d'un message.




```
public function finishQuestion() {
    // Supprimer toutes les réponses sauf la bonne
    for(var i:int=0;i<4;i++) {
        answerSprites.getChildAt(i).removeEventListener(MouseEvent.CLICK,clickAnswer);
        if (answers[i] != correctAnswer) {
            answerSprites.getChildAt(i).visible = false;
        } else {
            answerSprites.getChildAt(i).y = 200;
        }
    }
}
```

Le score doit aussi être mis à jour, ainsi que le pointeur `questionNum`. Pour finir, un nouveau bouton est créé avec l'étiquette "Continue" (voir Figure 10.4 également) :

```
    // Question suivante
    questionNum++;
    numQuestionsAsked++;
    showScore();
    showGameButton("Continue");
}
```

Le bouton créé par `clickAnswer` est un lien vers la question suivante. Lorsque le joueur clique dessus, nous appelons `pressGameButton`, qui lance la nouvelle question ou passe à l'écran `gameover`.

Fin de la partie

L'image `gameover` contient un bouton Play Again (rejouer) qui reconduit le joueur au début d'une nouvelle partie. Avant cela, elle doit cependant appeler `cleanUp` pour supprimer tous les restes de la partie terminée :

```
// Suppression des sprites
public function cleanUp() {
    removeChild(gameSprite);
    gameSprite = null;
    questionSprite = null;
    answerSprites = null;
    dataXML = null;
}
```

Le jeu peut maintenant être redémarré.

Ce jeu de quiz simple suffit pour les sites Web ou les produits simples qui ne requièrent qu'une solution très rudimentaire. Pour un jeu de questions/réponses très sophistiqué, nous devons ajouter bien d'autres fonctionnalités.

Quiz version Deluxe

Codes sources

http://flashgameu.com

A3GPU10_TriviaGameDeluxe.zip

Pour améliorer notre version du jeu actuelle, nous allons ajouter un certain nombre de fonctionnalités qui la rendront plus excitante, plus stimulante et plus amusante.

Pour commencer, le joueur doit avoir un temps limité pour répondre aux questions. La plupart des jeux de quiz sur ordinateur ou à la télévision imposent cette contrainte.

Ensuite, nous ajouterons un bouton d'indice afin que le joueur puisse obtenir s'il le souhaite une petite aide supplémentaire. Il existe deux types d'indices et nous les ajouterons tous les deux.

Ensuite, nous rendrons le jeu plus informatif en plaçant des informations supplémentaires après chaque question, ce qui contribuera à rendre le jeu plus ludique. Les informations viendront compléter ce que le joueur vient d'apprendre en répondant à la question.

Pour finir, nous allons relooker le système de score. Celui-ci doit tenir compte du temps pris pour répondre à la question et du fait que le joueur a réclamé ou non un indice.

Enfin, en guise de bonus supplémentaire, nous amènerons le quiz à lire un grand nombre de questions mais à en sélectionner dix au hasard à utiliser. Le questionnaire sera ainsi différent à chaque nouvelle partie.

Définir un temps limite

Pour ajouter une contrainte de temps au jeu, nous avons besoin d'une représentation visuelle du temps que le joueur a pour répondre à une question. Nous pouvons procéder en utilisant un objet clip séparé. Cet objet `Clock` peut correspondre à n'importe quel type d'outil représentant le temps : une horloge, du texte ou quoi que ce soit d'autre.

Pour cet exemple, j'ai configuré un clip de vingt-six images. Toutes les images contiennent vingt-cinq cercles. À partir de l'image 2, l'un des cercles est rempli avec une forme unie. Dans la première image, les vingt-cinq cercles sont donc vides. À la vingt-sixième, ils sont tous remplis. La Figure 10.5 présente ce clip `Clock`.

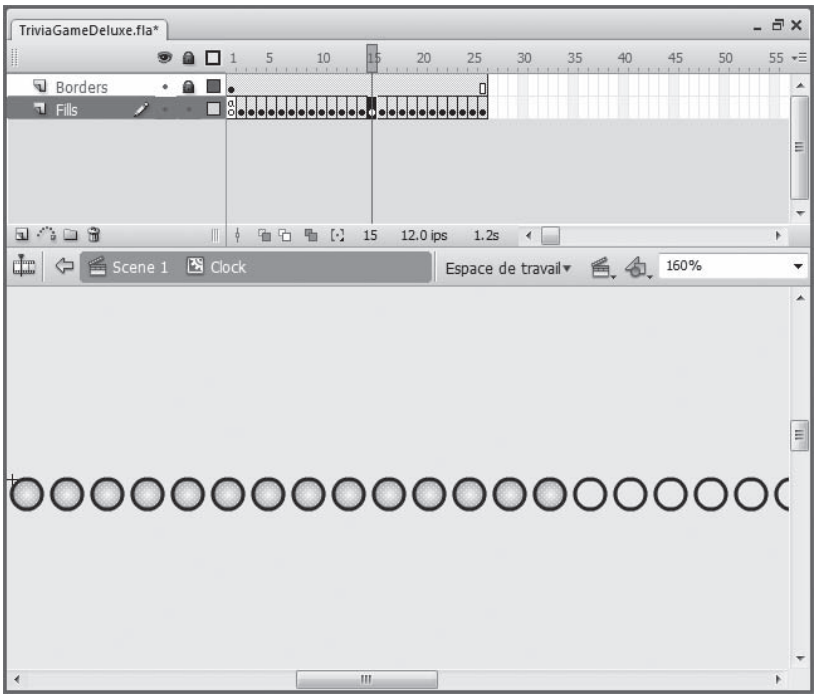


Figure 10.5
La quinzième image du clip Clock contient quatorze cercles remplis.

Nous utiliserons un `Timer` pour compter les secondes. Ajoutons donc une instruction `import` à cet effet :

```
import flash.utils.Timer;
```

Ensuite, nous ajoutons un `Clock` aux sprites utilisés :

```
private var clock:Clock;
```

Et un `Timer` :

```
private var questionTimer:Timer;
```

Dans la fonction `askQuestion`, nous devons ajouter le `Clock` et lancer le `Timer` :

```
// Configurer une nouvelle horloge
clock = new Clock();
clock.x = 27;
clock.y = 137.5;
questionSprite.addChild(clock);
questionTimer = new Timer(1000,25);
questionTimer.addEventListener(TimerEvent.TIMER,updateClock);
questionTimer.start();
```

L'objet `Clock` sera positionné juste en dessous de la question à l'écran. En fait, nous devons étendre un peu la hauteur du jeu et déplacer les éléments vers le bas pour faire place à notre objet `Clock` et à certains autres éléments que nous allons ajouter sous peu. La Figure 10.6 présente cette nouvelle disposition.

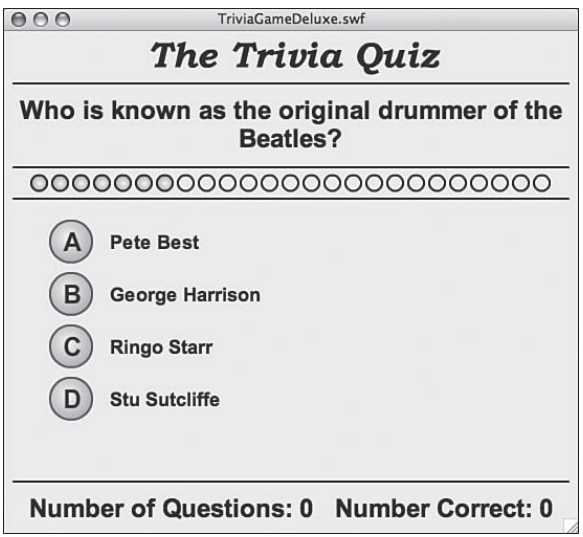


Figure 10.6
L'objet Clock a été ajouté et il reste de la place pour d'autres éléments.

L'utilisation de vingt-cinq points pour l'horloge est complètement arbitraire. Vous pourriez créer n'importe quelle séquence en vingt-six images sous forme de clip et l'utiliser (un chronomètre ou une barre de progression, par exemple). Il n'est même pas nécessaire d'utiliser vingt-cinq éléments séparés. Vous pourriez aisément remplacer le tout par cinq modifications et étendre les images le long du scénario.

La fonction `updateClock` est appelée à chaque seconde. Le clip `Clock` avance d'une image de plus. Lorsque le temps est écoulé, un message est affiché et une question finale (`finishQuestion`) est appelée exactement comme lorsque le joueur clique sur une réponse :

```
// Mettre à jour l'horloge
public function updateClock(event:TimerEvent) {
    clock.gotoAndStop(event.target.currentCount+1);
    if (event.target.currentCount == event.target.repeatCount) {
        messageField = createText("Out of time! The correct answer was:",
questionFormat,gameSprite,0,190,550);
        finishQuestion();
    }
}
```

À présent, le joueur a deux moyens d'obtenir une mauvaise réponse : en choisissant une proposition incorrecte ou en dépassant le délai autorisé.

Ajouter des indices

Vous aurez peut-être remarqué que les fichiers XML d'exemple incluent à la fois un indice (`hint`) et une info supplémentaire (`extra fact`) pour toutes les questions. Nous allons enfin nous en servir.

Pour ajouter des indices simples au jeu, nous inclurons un simple bouton "Hint" (indice) à côté de chaque question. Lorsque le joueur cliquera dessus, nous le remplacerons par le texte de l'indice.

Il nous faut quelques nouveaux éléments pour implémenter ce mécanisme. Tout d'abord, nous allons ajouter un `hintFormat` aux définitions de la classe, avec les définitions de variable texte :

```
private var hintFormat:TextFormat;
```

Ensuite, nous définirons ce format dans la fonction de construction :

```
hintFormat = new TextFormat("Arial",14,0x330000,true,false,false,null,null,"center");
```

Nous ajouterons également un `hintButton` à la liste des variables de la classe, avec les définitions de sprites et d'objets :

```
private var hintButton:GameButton;
```

Dans la fonction `askQuestion`, nous créerons le nouveau bouton Hint et le positionnerons en dessous de la dernière réponse, comme le montre la Figure 10.7 :

```
// Placer le bouton d'indice
hintButton = new GameButton();
hintButton.label.text = "Hint";
hintButton.x = 220;
hintButton.y = 390;
gameSprite.addChild(hintButton);
hintButton.addEventListener(MouseEvent.CLICK,pressedHintButton);
```

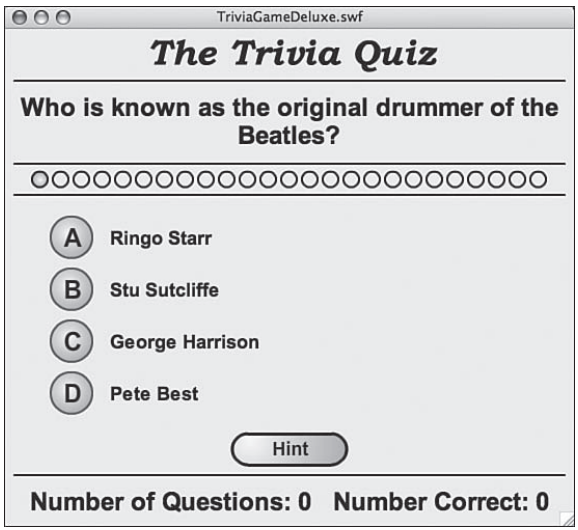


Figure 10.7
Le bouton Hint apparaît vers le bas.

Lorsque le joueur clique sur le bouton, celui-ci disparaît. Il est remplacé par un nouveau champ texte auquel est attribué le format de texte `hintFormat` :

```
// Le joueur veut un indice
public function pressedHintButton(event:MouseEvent) {
    // Supprimer le bouton
    gameSprite.removeChild(hintButton);
    hintButton = null;

    // Afficher l'indice
    var hint:String = dataXML.item[questionNum].hint;
    var hintField:TextField = createText(hint,hintFormat,questionSprite,0,390,550);
}
```

Nous souhaitons aussi utiliser l'instruction `removeChild` à l'intérieur de la fonction `finishQuestion`, en vérifiant d'abord que le `hintButton` existe, au cas où il aurait été supprimé lorsque le joueur a cliqué dessus :

```
//Supprimer bouton d'indice
if (hintButton != null) {
    gameSprite.removeChild(hintButton);
}
```

Cette vérification empêche que l'utilisateur clique sur le bouton après que la réponse à la question eut déjà été donnée.

Voilà tout ce qu'il nous faut pour afficher l'indice. Comme `hintField` fait partie de `questionSprite`, il est nettoyé lorsque nous supprimons ce sprite à la fin de la partie. La Figure 10.8 montre l'indice qui apparaît après que le joueur a cliqué sur le bouton.

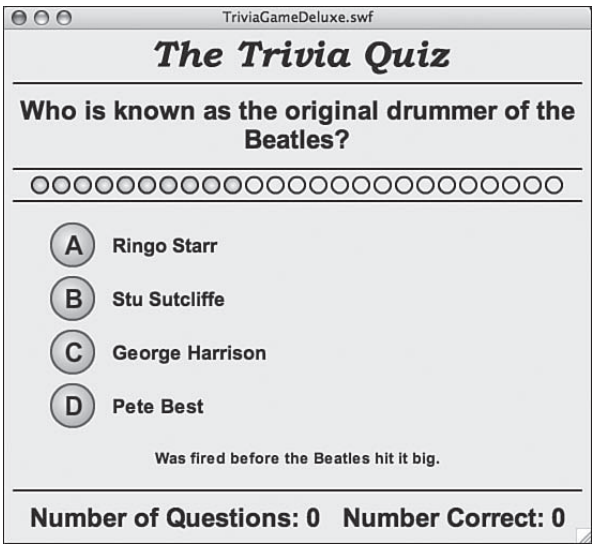


Figure 10.8
L'indice apparaît à la place du bouton

Qu'est-ce qu'un bon indice ? Il peut parfois être plus difficile de rédiger un bon indice que la question et les réponses. Il ne faut pas donner la réponse directement et, pourtant, il faut aider le joueur malgré tout. Souvent, le meilleur moyen consiste à donner un indice qui suggère la réponse mais dans un autre contexte. Par exemple, si la question concerne des noms de présidents et que la réponse est Charles de Gaulle, l'indice pourrait être "Il a donné son nom à un aéroport".

Ajouter des informations supplémentaires

Il est assez facile d'ajouter des informations supplémentaires à la fin d'une question. Cette fonctionnalité est sembable à celle des indices, à ceci près que les informations s'affichent automatiquement lorsque le joueur a répondu à la question.

Aucune nouvelle variable n'est requise pour cela. Il nous suffit de créer un champ texte et de le remplir lorsque la question est terminée. Le code suivant est ajouté à `finishQuestion` :

```
// Afficher les informations supplémentaires
var fact:String = dataXML.item[questionNum].fact;
var factField:TextField = createText(fact,hintFormat,questionSprite,0,340,550);
```

Comme le nouveau `TextField` fait partie de `questionSprite`, il est supprimé en même temps. Nous utilisons également le format `hintFormat` au lieu de créer un format séparé pour ces informations. La Figure 10.9 présente le résultat.

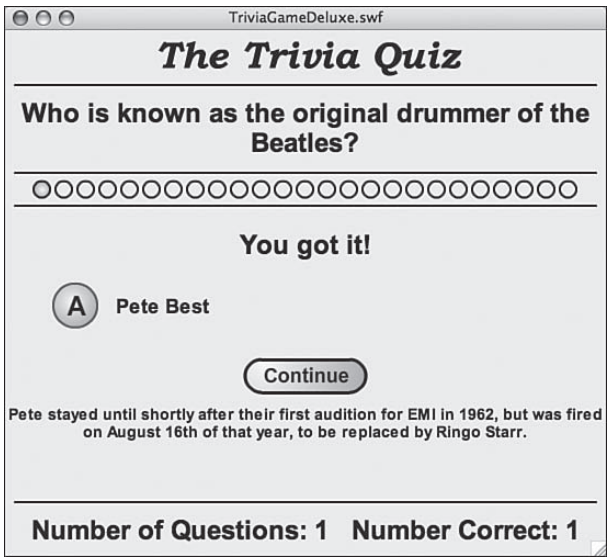


Figure 10.9
Les informations supplémentaires s'affichent lorsque l'utilisateur a répondu à la question.

Au moment de décider où placer les informations supplémentaires, je me suis assuré que l'indice et les informations pouvaient cohabiter à l'écran. Si le joueur choisit d'afficher l'indice, il reste à l'écran après que le joueur a répondu à la question et apparaît juste en dessous des informations supplémentaires.

Ajouter un système de score complexe

Le problème avec la fonctionnalité des indices ainsi qu'avec l'horloge tient à ce que le joueur est très peu pénalisé par le fait de profiter de l'indice ou de laisser s'écouler un long délai avant de répondre.

Ce qui rend le jeu plus excitant, c'est d'avoir une pénalité en cas d'usage de l'indice. En outre, nous pouvons comptabiliser le nombre de points marqués en fonction de la rapidité avec laquelle le joueur répond à la question.

Pour opérer ces modifications, introduisons deux nouvelles variables. Elles peuvent être placées n'importe où dans les définitions de variables, même s'il est préférable de les placer avec les définitions de variables de l'état du jeu existantes. Elles tiendront le registre du nombre de points que vaut la question actuelle et du nombre de points total que le joueur a marqués jusque-là dans la partie :

```
private var questionPoints:int;
private var gameScore:int;
```

Dans la fonction `startTriviaGame`, nous initialiserons `gameScore` à `0`, juste avant d'appeler `showScore` :

```
gameScore = 0;
```

La fonction `showScore` sera remplacée par une nouvelle version. Celle-ci affichera le nombre de points que vaut la question et le score actuel du joueur :

```
public function showScore() {
    if (questionPoints != 0) {
        scoreField.text = "Potential Points: "+questionPoints+"\t   Score: "+gameScore;
    } else {
        scoreField.text = "Potential Points: ---\t   Score: "+gameScore;
    }
}
```

Les caractères `\t` dans la valeur de `scoreField.text` représentent un caractère de tabulation. En plaçant une tabulation entre les deux parties du champ, nous donnons la possibilité au texte de rester dans le même espace général, même lorsque la longueur des nombres change. Ce n'est pas une solution parfaite, mais elle est bien plus simple que celle qui consiste à créer deux champs séparés dans ce cas. Vous pourriez utiliser deux champs séparés si vous souhaitez mieux contrôler le placement de ces nombres.

La Figure 10.10 montre comment le nouveau score est affiché en bas de l'écran.

Maintenant que la fonction `showScore` doit mettre à jour les points potentiels et le score total, nous devons l'appeler plus souvent. À chaque fois que le `questionScore` change, nous devons appeler `showScore` pour faire connaître au joueur la nouvelle valeur.

Si `questionScore` vaut `0`, nous affichons `---` au lieu de `0`. Nous indiquerons ainsi plus clairement que les points potentiels ne signifient rien entre les questions.

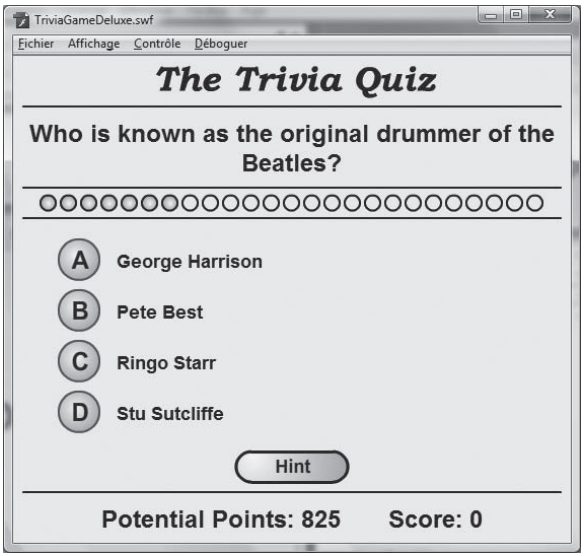


Figure 10.10
Le nombre de questions posé et le nombre de bonnes réponses ont été remplacés par le nombre de points potentiel pour la question et le score actuel du joueur.

Dans `askQuestion`, nous fixerons le score potentiel de la question à 1 000 :

```
// Commencer la question avec les points au maximum
questionPoints = 1000;
showScore();
)
```

À chaque seconde qui passe, nous diminuons ensuite le score. Cette procédure intervient dans la fonction `updateClock`. À chaque fois qu'un nouveau cercle est rempli, nous supprimons 25 points pour le score potentiel :

```
// Réduction des points
questionPoints -= 25;
showScore();
```

Les points potentiels diminuent aussi lorsque le joueur demande à afficher l'indice. Cette opération lui coûte 300 points :

```
// Pénalité
questionPoints -= 300;
showScore();
```

Bien évidemment, le seul moyen d'obtenir des points consiste à donner la bonne réponse. L'instruction suivante est donc ajoutée à l'emplacement adéquat dans `clickAnswer` :

```
gameScore += questionPoints;
```

Inutile ici d'appeler `showScore` car la fonction sera appelée immédiatement après dans la fonction `finishQuestion`. C'est d'ailleurs à cet endroit que nous allons positionner `questionPoints` à `0` également :

```
questionPoints = 0;
showScore();
```

Vous pouvez aussi choisir de conserver le champ texte d'oirigine du score en bas et afficher les points potentiels et le score dans un champ séparé. Les joueurs pourront alors voir toutes les statistiques concernant leur résultat.

Les animations **TriviaGameDeluxe.fla** et **TriviaGameDeluxe.as** conservent `numCorrect` et `numQuestionsAsked` à cet effet, bien qu'elles ne les utilisent pas.

Rendre les questions aléatoires

Vous pourrez souhaiter ou non que votre jeu de quiz présente les mêmes questions à chaque nouvelle partie. Tout dépend de votre manière d'utiliser ce jeu.

Si vous souhaitez présenter des questions différentes à chaque fois et que votre jeu est mis à disposition sur un site Web, l'idéal est d'avoir une application serveur qui crée un document XML aléatoire de questions et réponses à partir d'une grande base de données.

Si vos besoins sont plus simples mais que vous souhaitiez néanmoins qu'un certain nombre de questions aléatoires soient choisies parmi un nombre total plutôt restreint de questions, il existe un moyen de procéder en ActionScript.

Une fois le document XML lu, ces données brutes peuvent être traitées pour aboutir à un document XML plus petit ne contenant qu'un nombre défini de questions aléatoires.

Le nouveau début de la fonction `xmlLoaded` ressemble alors à ceci :

```
public function xmlLoaded(event:Event) {
    var tempXML:XML = XML(event.target.data);
    dataXML = selectQuestions(tempXML,10);
```

La fonction `selectQuestions` prend en paramètres le jeu de données complet et un nombre de questions à retourner. Elle sélectionne des nœuds `item` aléatoires du document XML d'origine et crée un nouvel objet XML :

```
// Sélectionner un nombre donné de questions aléatoires
public function selectQuestions(allXML:XML, numToChoose:int):XML {

    // Créer un nouvel objet XML pour contenir les questions
    var chosenXML:XML = <trivia></trivia>;
```

```
        // Boucler jusqu'à ce que nous ayons assez de questions
        while(chosenXML.child("*").length() < numToChoose) {

            // Sélectionner une question au hasard et la déplacer
            var r:int = Math.floor(Math.random()*allXML.child("*").length());
            chosenXML.appendChild(allXML.item[r].copy());

            // Ne pas utiliser à nouveau
            delete allXML.item[r];
        }

        // Retour
        return chosenXML;
    }
```

Cette sélection aléatoire et ce mélange des questions sont très pratiques pour créer une solution rapide. Si vous avez par exemple plus de cent questions, il est en revanche important de ne pas demander à l'animation de lire un document XML aussi grand en entier à chaque fois. Je recommande la solution côté serveur. Si vous n'avez pas l'habitude de programmer des solutions côté serveur, associez-vous avec quelqu'un d'aguerri dans cet exercice.

Quiz d'images

Codes sources
http://flashgameu.com
A3GPU10_PictureTriviaGame.zip

Tous les jeux de questions et de réponses ne fonctionnent pas nécessairement avec du texte. Il arrive qu'une image soit mieux à même de représenter une idée. Par exemple, si vous souhaitez tester les connaissances d'une personne en géométrie, les questions et réponses sous forme de texte ne sont pas toujours les mieux adaptées pour opérer des évaluations.

Il n'est en fait pas si difficile de convertir notre moteur de jeu de quiz en une nouvelle version qui utilise des images. Il suffit de réorganiser légèrement l'écran puis de permettre le chargement de fichiers d'image externes. L'essentiel du jeu peut être conservé en l'état.

Améliorer la disposition des réponses

Avant de pouvoir charger des images, nous devons améliorer la disposition des réponses à l'écran. La Figure 10.11 présente les réponses selon un quadrillage de 2 × 2 au lieu de quatre lignes.

Nous disposons ainsi d'un meilleur cadre pour des images d'environ 250 pixels de large et 100 de haut au maximum. Il sera préférable de s'en tenir à 200 × 80 afin que les images chargées ne viennent pas gêner les autres boutons.

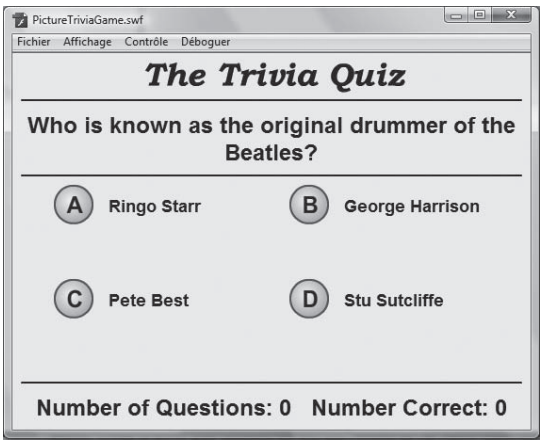


Figure 10.11
Les réponses sont maintenant empilées dans deux colonnes et deux lignes.

Cette disposition peut être obtenue en n'opérant qu'un changement au milieu de la fonction `askQuestion`. Les variables `xpos` et `ypos` tiennent le registre de la position courante et commencent à 0 et 0. Ensuite, 1 est ajouté à `xpos` afin de passer vers la droite. Ensuite, `xpos` est ramenée à `0` et `ypos`, augmentée. Cette procédure place les quatre réponses aux positions (0,0), (1,0), (0,1) et (1,1), ce qui correspond aux emplacements (100,150), (350,150), (100,250) et (350,250) à l'écran.

*Nous allons apporter d'autres modifications à cette section de code sous peu. Le code qui suit ne correspondra donc pas au fichier **PictureTriviaGame.as** si vous avancez parallèlement au livre.*

```
// Placer chaque réponse dans un nouveau sprite avec une icône de cercle
answerSprites = new Sprite();
var xpos:int = 0;
var ypos:int = 0;
for(var i:int=0;i<answers.length;i++) {
```

```
        var answer:String = answers[i];
        var answerSprite:Sprite = new Sprite();
        var letter:String = String.fromCharCode(65+i); // A-D
        var answerField:TextField = createText(answer,answerFormat,answerSprite,0,0,200);
        var circle:Circle = new Circle(); // De la bibliothèque
        circle.letter.text = letter;
        answerSprite.x = 100+xpos*250;
        answerSprite.y = 150+ypos*100;
        xpos++
        if (xpos > 1) {
            xpos = 0;
            ypos += 1;
        }
        answerSprite.addChild(circle);
        answerSprite.addEventListener(MouseEvent.CLICK,clickAnswer); // En faire un bouton
        answerSprites.addChild(answerSprite);
    }
```

Cette modification est déjà utile parce qu'elle présente les réponses d'une manière plus intéressante qu'en les empilant les unes au-dessus des autres.

Reconnaître deux types de réponses

Le but ici est non pas de créer un quiz qui ne propose que des images comme réponse, mais de vous permettre de combiner du texte et des images. Nous devons donc pouvoir spécifier dans le fichier XML le type de la réponse. Cela peut être fait en ajoutant un attribut à la réponse dans le code XML :

```
<item>
    <question type="text">Which one is an equilateral triangle?</question>
    <answers>
        <answer type="file">equilateral.swf</answer>
        <answer type="file">right.swf</answer>
        <answer type="file">isosceles.swf</answer>
        <answer type="file">scalene.swf</answer>
    </answers>
</item>
```

Pour déterminer si une réponse doit être affichée sous forme de texte ou de fichier externe chargé, nous examinons simplement la propriété `type`. Nous allons maintenant modifier notre code pour cela.

Créer des objets *Loader*

Dans `shuffleAnswers`, nous créons un tableau de réponses trié aléatoirement à partir des réponses dans l'objet XML. Le texte de ces réponses est stocké dans un tableau. Nous devons cependant maintenant stocker à la fois le texte et le type de ces réponses. La ligne où nous ajoutons une nouvelle réponse au tableau prend ainsi la forme suivante :

```
shuffledAnswers.push({type: answers.answer[r].@type, value: answers.answer[r]});
```

À présent, lorsque nous créons chaque réponse, nous devons déterminer si la réponse correspond à du texte ou à une image. S'il s'agit d'une image, nous créons un objet `Loader`. Celui-ci agit comme un clip récupéré de la bibliothèque, sauf que l'on utilise un `URLRequest` et la commande `load` pour récupérer le contenu du clip à partir d'un fichier externe :

```
var answerSprite:Sprite = new Sprite();
if (answers[i].type == "text") {
    var answerField:TextField = createText(answers[i].value,answerFormat,answerSprite,
     0,0,200);
} else {
    var answerLoader:Loader = new Loader();
    var answerRequest:URLRequest = new URLRequest("triviaimages/"+answers[i].value);
    answerLoader.load(answerRequest);
    answerSprite.addChild(answerLoader);
}
```

Le code présuppose que toutes les images se trouvent à l'intérieur d'un dossier nommé `triviaimages`.

Les objets `Loader` peuvent agir de manière autonome. Une fois que vous les activez avec la commande `load`, ils obtiennent le fichier depuis le serveur et apparaissent à leur position désignée lorsqu'ils sont prêts. Vous n'avez pas besoin de les surveiller ni de faire quoi que ce soit lorsque le chargement est terminé.

Si vous combinez cet exemple avec la fonction `Clock` de l'animation précédente, vous aurez un peu de travail supplémentaire. Ne serait-il pas injuste que le chronomètre se mette en marche alors que certaines des réponses ne sont pas encore apparues ? Il convient donc d'écouter `Event.COMPLETE` pour chaque `Loader` et de ne lancer le chronomètre qu'une fois que toutes les réponses ont été affichées.

La Figure 10.12 présente le quiz avec quatre animations externes chargées dans les réponses.

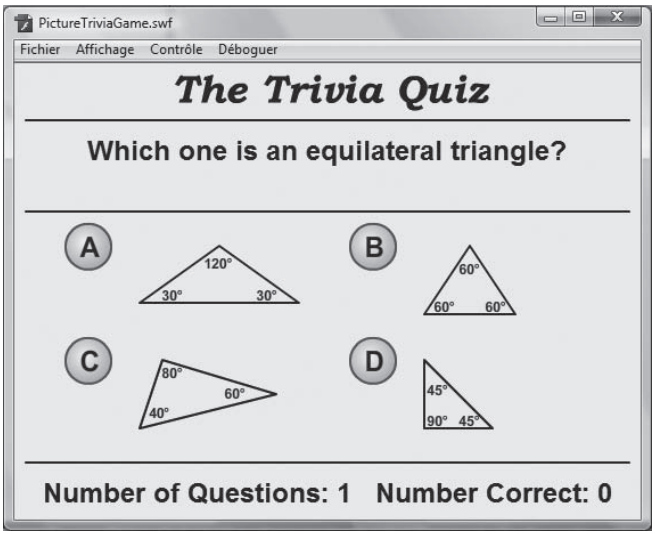


Figure 10.12
Des animations externes ont remplacé le texte dans chaque réponse.

Déterminer la bonne réponse

Nous nous sommes auparavant appuyés sur la propriété `text` du champ de la réponse pour déterminer si le joueur avait choisi la bonne réponse. Nous ne pouvons plus le faire à présent car le clip de l'objet `Loader` ne possède pas de propriété `text` comme les objets `TextField`. Au lieu de cela, nous allons donc tirer parti du fait que le second objet dans `answerSprite` correspond à l'objet `Circle` créé de manière dynamique. Nous pouvons lui attacher une propriété `answer` et y stocker la réponse :

```
circle.answer = answers[i].value;
```

Ensuite, dans la fonction `clickAnswer`, nous examinerons cette nouvelle propriété `answer` afin de déterminer si le joueur a cliqué sur le bon sprite :

```
var selectedAnswer = event.currentTarget.getChildAt(1).answer;
```

Vous remarquerez que l'objet `Circle` est l'enfant numéro 1 dans `answerSprite`. Auparavant, nous examinions l'enfant numéro 0, qui était l'objet `TextField`.

Une autre modification est requise afin de définir correctement la position de la bonne réponse lorsque le joueur a opéré son choix. Auparavant, les réponses se trouvaient toutes dans une unique colonne, avec la même valeur `x`. Lorsque nous souhaitions centrer la bonne réponse, nous n'avions donc qu'à définir la valeur `y` du bon `answerSprite`. Maintenant que la réponse peut se trouver à gauche ou à droite, nous devons aussi positionner la valeur `x`. Voici le nouveau code pour la fonction `finishQuestion` :

```
answerSprites.getChildAt(i).x = 100;
answerSprites.getChildAt(i).y = 200;
```

Étendre la zone réactive

Un dernier point avant que nous n'en ayons fini avec les réponses. Si la réponse est une réponse texte, les joueurs peuvent cliquer sur le cercle ou le champ texte pour enregistrer leur réponse. Avec les animations chargées, il peut ne pas y avoir grand-chose à cliquer. À la Figure 10.11, les réponses ne sont rien d'autre que des lignes fines qui composent un triangle.

Pour cliquer sur la réponse, le joueur doit alors cliquer sur le cercle ou sur une partie du graphisme. Il faudrait pourtant qu'il puisse cliquer sur n'importe quelle partie normale de la réponse.

L'un des moyens rapides de résoudre ce problème consiste à placer un rectangle plein à l'intérieur de chaque sprite de réponse. Il suffit pour cela de tracer un rectangle plein en fixant à 0 le canal alpha afin de le rendre invisible :

```
// set a larger click area
answerSprite.graphics.beginFill(0xFFFFFF,0);
answerSprite.graphics.drawRect(-50, 0, 200, 80);
```

La Figure 10.13 présente les sprites de réponse avec un graphisme sous-jacent. Au lieu de rendre la valeur nulle, j'ai fixé la valeur alpha à 0,5 afin que le rectangle se remarque.

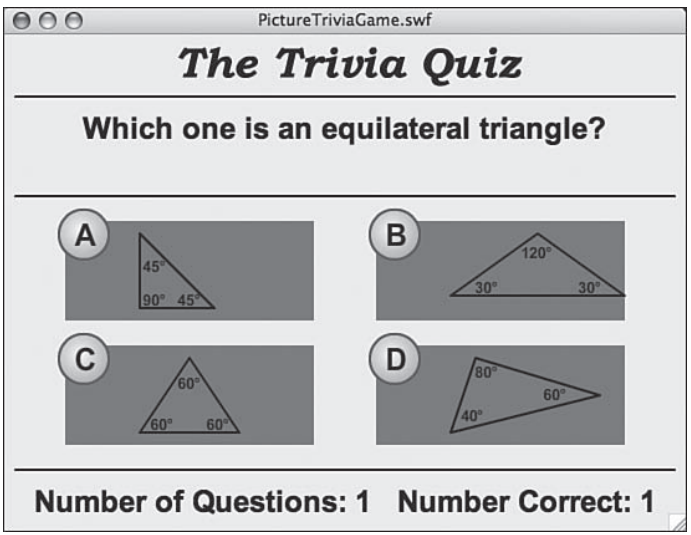


Figure 10.13
Un rectangle a été dessiné derrière chaque réponse.

Le joueur peut maintenant cliquer dans la zone complète de chaque réponse.

Images pour les questions

En plus d'utiliser des images dans les réponses, vous pourriez souhaiter utiliser des images pour la question elle-même. Nous procéderons de la même manière, en utilisant un attribut `type` du code XML :

```
<item>
    <question type="file">italy.swf</question>
    <answers>
        <answer type="text">Italy</answer>
        <answer type="text">France</answer>
        <answer type="text">Greece</answer>
        <answer type="text">Fenwick</answer>
    </answers>
</item>
```

Cet ajout à notre code ActionScript est plus simple car la question n'est pas cette fois un élément actif. Nous avons simplement besoin d'utiliser un objet `Loader` au lieu d'un objet `TextField`. Voici le changement apporté à `askQuestion` :

```
// Créer un champ texte pour la question
var question:String = dataXML.item[questionNum].question;
if (dataXML.item[questionNum].question.@type == "text") {
    questionField = createText(question,questionFormat,questionSprite,0,60,550);
} else {
    var questionLoader:Loader = new Loader();
    var questionRequest:URLRequest = new URLRequest("triviaimages/"+question);
    questionLoader.load(questionRequest);
    questionLoader.y = 50;
    questionSprite.addChild(questionLoader);
}
```

La Figure 10.14 présente une question qui utilise une image externe comme question et quatre réponses texte. Vous pourriez évidemment utiliser des images externes aussi bien pour la question que pour les quatre réponses.

La Figure 10.14 montre comment le fait d'utiliser des fichiers externes pour les questions et les réponses n'implique pas nécessairement qu'il s'agisse de dessins ou d'images. Il peut aussi y avoir du texte. Ce mélange peut se révéler pratique pour un texte de mathématiques qui doit utiliser des notations complexes comme des fractions, des puissances ou divers symboles.

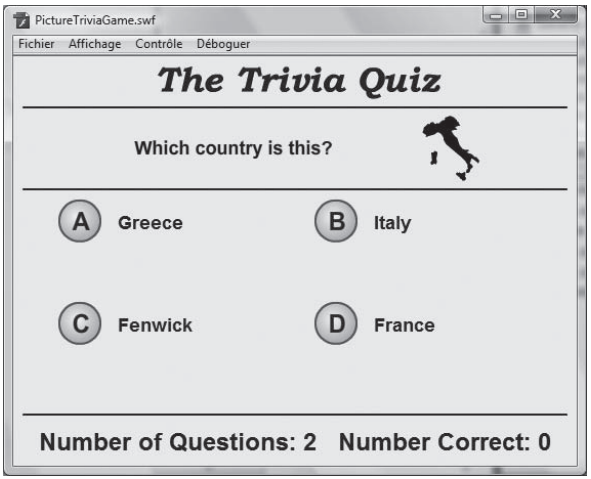


Figure 10.14
La question est une animation Flash externe, mais les quatre réponses sont du texte.

Modifier le jeu

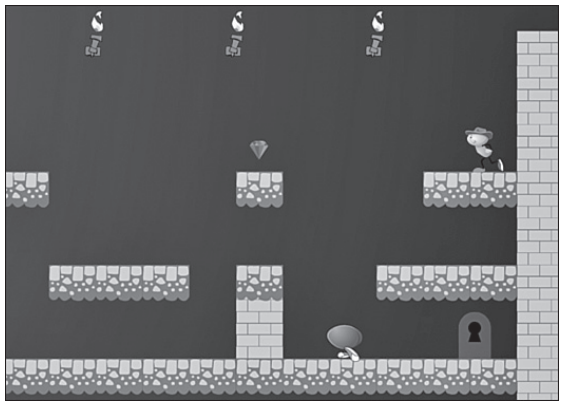
La qualité des jeux de quiz dépend de celle des questions et des réponses qu'ils contiennent, quelle que soit par ailleurs la qualité du programme et de l'interface. Si vous prévoyez de créer un jeu à des fins de divertissement, il vous faut des questions et des réponses suffisamment attrayantes pour susciter l'intérêt des joueurs. Si vous créez un jeu à des fins éducatives, vous devez vous assurer que les questions et les réponses soient claires et correctement formulées.

Il est assez aisé de modifier ce jeu de manière à inclure un plus grand nombre ou un plus petit nombre de réponses. Si vous le souhaitez, vous pouvez ne proposer que deux réponses, comme Vrai et Faux. Les questionnaires à choix multiples proposent cependant rarement plus de quatre réponses, bien que l'on voie parfois des propositions du type "Toutes" ou "Aucune" pour valider ou invalider toutes les réponses précédentes. Aucune programmation particulière n'est requise pour créer une solution de ce type, sauf éventuellement pour s'assurer que ces réponses globales soient bien proposées en cinquième ou sixième choix dans la liste.

Au-delà des questions et des réponses et de la manière dont elles sont affichées, l'une des modifications possibles pourrait être de proposer une métaphore du jeu. Il s'agirait d'une représentation visuelle du score du joueur qui pourrait aussi modifier la manière de jouer le jeu.

Par exemple, le joueur pourrait avoir un personnage qui grime à une corde. À chaque bonne réponse, le personnage grimperait d'un cran. À chaque mauvaise réponse, il retomberait en bas. Le but serait de parvenir en haut en répondant correctement à un certain nombre de questions de suite.

Les métaphores de jeu peuvent être utilisées pour mieux harmoniser le jeu avec le site Web ou le produit dont il fait partie. Par exemple, un site de préservation de la nature pourrait proposer un jeu de questions/réponses qui utilise des animaux.



11

Jeux d'action : jeux de plate-forme

Au sommaire de ce chapitre :

- Concevoir le jeu
- Créer la classe
- Modifier le jeu

Codes sources

http://flashgameu.com

A3GPU11_PlatformGame.zip

Les jeux à défilement latéral ou jeux de plate-forme sont apparus pour la première fois au début des années 1980 et sont rapidement devenus un standard des jeux vidéo jusqu'à ce que la 3D ne s'impose dans le courant des années 1990.

Les jeux de plate-forme à défilement permettent au joueur de contrôler un personnage qui apparaît de profil. Il peut se déplacer vers la gauche et vers la droite ainsi que sauter le plus souvent. Lorsqu'il avance vers un côté de l'écran, l'arrière-plan défile afin de révéler d'autres portions du jeu.

Les jeux de plate-forme incluent presque toujours un personnage capable de sauter. Le plus célèbre d'entre eux est évidemment Mario, de Nintendo. Il apparaît dans des dizaines de jeux, de Donkey Kong jusqu'à de nombreux jeux d'aventure des consoles Nintendo.

Dans ce chapitre, nous allons créer un jeu de plate-forme avec un personnage principal qui se déplace vers la gauche et vers la droite et peut sauter. Nous inclurons des murs et des plates-formes. Il y aura aussi des ennemis et des objets à ramasser.

Conception du jeu

Avant de commencer à programmer notre code, commençons pas considérer tous les aspects du jeu. Il nous faut un héros, des ennemis, des objets et un moyen de créer des niveaux.

Conception des niveaux

L'un des aspects importants des jeux de plate-forme concerne la conception des niveaux. Tout comme le jeu de quiz du précédent chapitre ne peut exister sans contenu, le jeu de plate-forme requiert aussi un contenu. Il s'agit en l'occurrence d'une série de niveaux.

Quelqu'un (le programmeur, un graphiste ou un concepteur de niveau) doit créer les niveaux. La Figure 11.1 présente un niveau qui sera en fait le premier de notre jeu.

Les jeux de plate-forme plus sophistiqués peuvent aussi intégrer un éditeur de niveau qui permet au concepteur de niveau de créer de nombreux niveaux et de les tester pendant que les programmeurs et les graphistes travaillent sur le reste du jeu.

Pour notre exemple simple, nous concevrons cependant les niveaux directement dans Flash, en utilisant un clip contenant les différentes pièces du jeu.

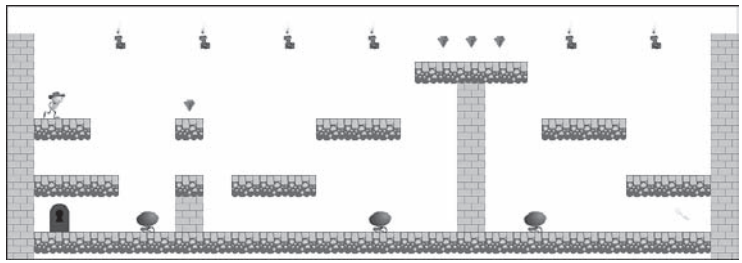


Figure 11.1
Le niveau 1 de notre jeu de plate-forme inclut trois ennemis, plusieurs trésors, une clé et une porte.

L'animation **PlatformGame.fla** contient différentes pièces de jeu dans sa bibliothèque. En voici la liste :

- **Sol.** Un bloc de 40 × 40 simple sur lequel le héros peut tenir debout. Il bloquera le héros côté gauche et côté droit.
- **Mur.** Identique au sol quant à l'usage, mais visuellement différent.
- **Héros.** Le personnage du joueur. Inclut des animations pour la position debout, la marche, le saut et la mort.
- **Ennemi.** Un personnage plus simple avec une animation de marche.
- **Trésor.** Un bijou simple, avec des reflets animés.
- **Clé.** Un graphisme de clé simple.
- **Porte.** Une porte, avec une animation pour l'ouverture.

Pour créer un niveau à partir de ces objets, il suffit de les positionner à l'intérieur d'un clip. La Figure 11.1 en donne un exemple. Il s'agit en fait du clip `gamelevel1` dans la bibliothèque de l'animation **PlatformGame.fla**.

Pièces des murs et du sol

Pour créer facilement ce niveau, j'ai mis en place une grille en choisissant Affichage > Grille > Modifier la grille et en définissant un quadrillage de 40 × 40. Ensuite, j'ai fait glisser les pièces du sol et des murs de la bibliothèque afin de les placer le long de cette grille.

La Figure 11.2 présente le niveau en masquant l'arrière-plan afin de révéler les repères de la grille. $

Les pièces des murs et du sol diffèrent visuellement dans ce jeu, mais seront considérés comme le même type d'objet dans le code. Vous pouvez ajouter d'autres "blocs constructeurs" de ce type pour mieux diversifier le jeu.

Parmi les idées qui ne sont pas utilisées dans ce jeu, l'une consiste à utiliser de nombreuses variantes pour les blocs des murs et du sol, en les stockant toutes dans différentes images du clip. Au début du jeu, une image aléatoire peut ainsi être choisie pour chaque bloc.

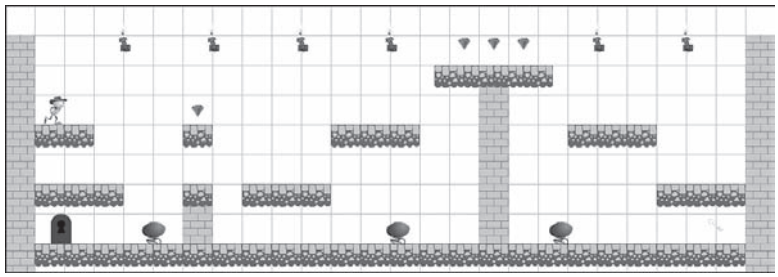


Figure 11.2
Le niveau peut être configuré plus facilement en plaçant les pièces des murs et du sol le long de la grille.

Les murs et le sol n'ont pas besoin de se voir attribuer un nom particulier. Les éléments de la bibliothèque sont néanmoins configurés de manière à être exportés pour ActionScript, comme le montre la Figure 11.3. Ce réglage est en revanche vital parce que notre code recherchera des objets `Floor` et `Wall`.

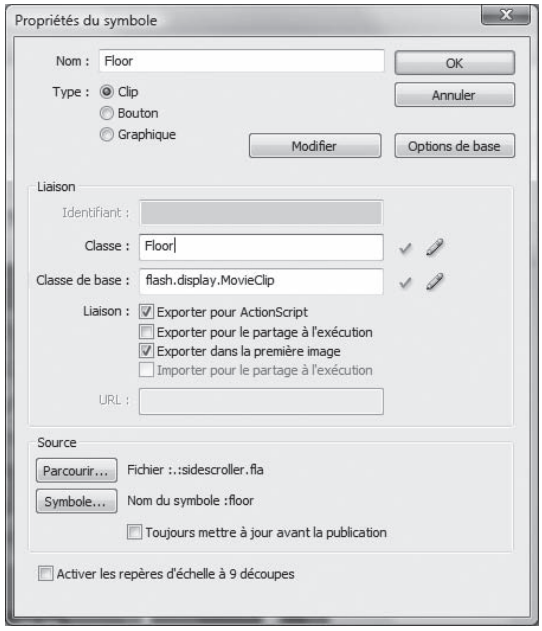


Figure 11.3
Le clip Floor est configuré pour être aussi l'objet de classe Floor.

Le héros et les ennemis

Le héros apparaît dans le coin supérieur gauche des Figures 11.1 et 11.2. Il a été soigneusement placé de manière à se tenir pile sur un bloc de sol (`Floor`).

Le clip du héros contient plusieurs sections animées différentes. Examinez la Figure 11.4.

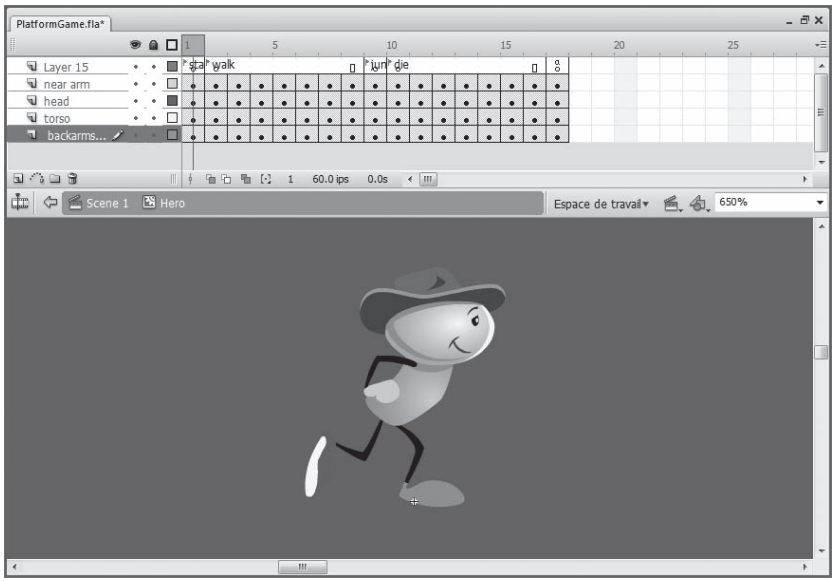


Figure 11.4
Le héros possède des images pour la position debout, la marche, le saut et la mort.

Vous remarquerez que le point d'alignement du clip se trouve exactement au bas des pieds du personnage. Nous le ferons correspondre au haut de l'élément `Floor`, sur lequel repose le personnage. Horizontalement, le personnage sera centré.

Au moment de placer le héros, il faudra que sa position `y` corresponde à la position `y` de l'élément `Floor` situé directement sous lui. Le héros commencera donc en position debout. Si vous le placez en hauteur au-dessus du bloc Floor, il commencera par tomber sur le sol. C'est une autre approche possible, d'ailleurs : le personnage pourrait ainsi paraître tomber au milieu de ce décor.

L'ennemi correspond à un clip similaire, avec une simple séquence de marche et une position debout (voir Figure 11.5). Vous remarquerez que le point d'alignement de l'ennemi se trouve également au bas de son pied.

Ces petits bonhommes sont conçus pour que l'on saute sur eux. C'est ainsi que le héros pourra s'en débarrasser. Ils sont donc courts sur pattes et comme accroupis. Nous n'aurons pas besoin d'une séquence de mort, car nous les supprimerons de l'écran dès l'instant où ils seront détruits et utiliserons l'explosion de points (la classe `PointBurst`) du Chapitre 8 pour afficher un message à cet endroit.

Les ennemis doivent aussi être placés directement sur un bloc `Floor`. Sinon ils tomberont sur le bloc `Floor` suivant : là encore, ce peut être un effet souhaité si vous le préférez.

Les ennemis requièrent aussi des noms d'occurrence. Les trois ennemis présentés dans les Figures 11.1 et 11.2 sont appelés `enemy1`, `enemy2` et `enemy3`.

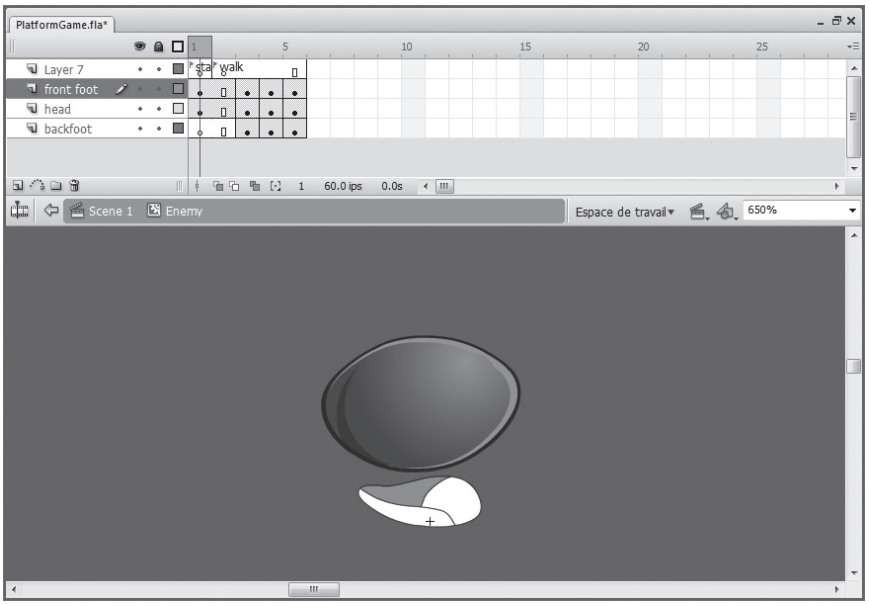


Figure 11.5
L'ennemi n'a que deux séquences pour la position debout et la marche.

Autre point concernant les ennemis : ils seront programmés pour marcher vers l'avant et l'arrière, en se tournant dès qu'ils rentrent dans un mur. Vous devez donc les placer dans une zone où figurent des murs des deux côtés. S'ils parviennent à une extrémité bordée par le vide, ils tomberont à la première occasion. Ils continueront de tomber ainsi à chaque fois qu'ils en auront la possibilité jusqu'à se stabiliser en avant et en arrière dans une zone bordée de murs des deux côtés.

Joyaux et objets

Différents objets apparaissent aux Figures 11.1 et 11.2. Les objets en forme de diamants sont des joyaux qui peuvent être ramassés pour gagner des points. Il n'y a rien de particulièrement remarquable concernant le clip `Treasure`, si ce n'est qu'il contient plusieurs images pour animer un effet d'éclat lumineux. Cette animation n'affecte en rien le jeu ; elle n'a qu'un intérêt visuel.

Si vous souhaitez plusieurs types de joyaux, le moyen le plus simple consiste à en placer un par image dans ce clip. Vous pouvez sinon créer une variété d'objets différents, comme des diamants, des pièces, des bagues, etc. Vous devrez alors les rechercher chacun dans le code.

Les clips `Key` et `Door` sont similaires. Le clip `Key` contient des images pour l'animation uniquement. Le clip `Door` contient une séquence d'ouverture qui démarre à l'image 2 (voir Figure 11.6).

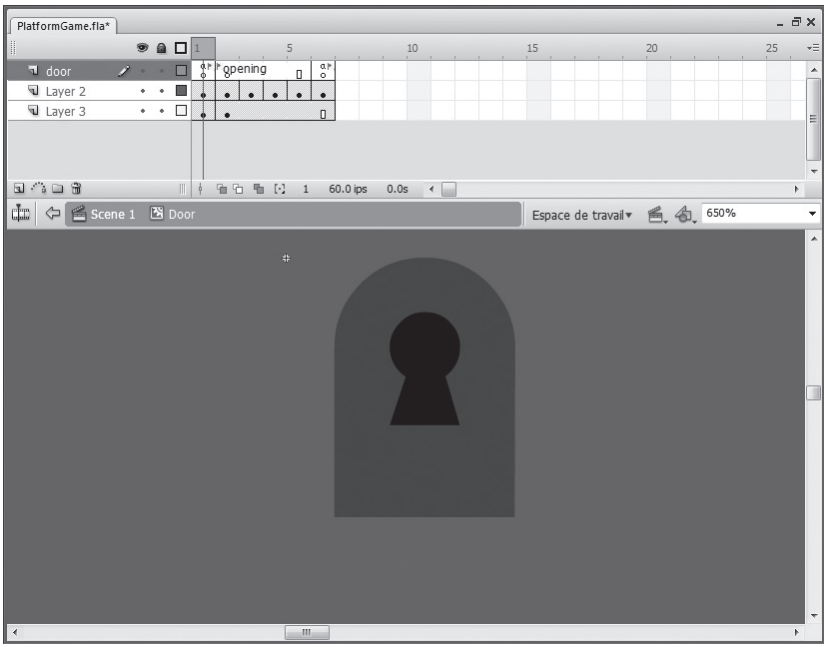


Figure 11.6
Le clip Door contient une image 1 statique et une courte séquence animée de son ouverture.

Les éléments n'ont pas besoin d'être parfaitement placés dans la grille. Ils doivent simplement être à portée du héros lorsqu'il marche ou dans d'autres cas, lorsqu'il saute. Visuellement, il est cependant souvent plus naturel qu'ils reposent sur le sol.

Faites attention à la superposition par calques de vos éléments de jeu. Elle sera conservée pendant le déroulement du jeu. Dès lors, si le héros se trouve derrière un mur ou un autre objet, ce mur apparaîtra devant le graphisme du héros lorsque ce dernier s'en rapproche.
Vous pouvez cependant volontairement faire apparaître des objets devant le joueur, comme une vitre semi-transparente ou un petit meuble.

Les clips `Treasure`, `Key` et `Door` sont tous configurés de manière à être exportés pour ActionScript avec des noms de classe correspondant à leurs noms de bibliothèque. Notre code les recherchera d'après la classe. Les occurrences des clips elles-mêmes ne doivent pas nécessairement avoir de noms.

Le coffre (`Chest`) fait partie des autres éléments du jeu. Ce clip à deux images présente un coffre de trésor fermé, puis ouvert. C'est l'objet de la quête du joueur. Le jeu se termine lorsque le joueur le trouve.

Graphismes d'arrière-plan

Les niveaux du jeu incluent également un clip avec des graphismes d'arrière-plan. Dans le cas présent, il s'agit d'un rectangle en dégradé. Il pourrait y avoir plus ; à vrai dire, tout ce que vous ajouterez à l'arrière-plan sera visible, sans être actif.

Vous pouvez donc colorier la scène comme vous le souhaitez. De grands tableaux peuvent être accrochés au mur ou des torches animées, des messages et des signes.

Aux Figures 11.1 et 11.2, des torches sont accrochées tout en haut. Elles se trouvent simplement placées sur le même calque d'arrière-plan que le dégradé. Le code de notre jeu n'a même pas besoin de les prendre en compte car elles ne feront que défiler avec l'arrière-plan.

La boîte de dialogue

Cette animation inclura une boîte de dialogue (voir Figure 11.7) que nous pourrons afficher à tout moment pour présenter des informations et attendre la réponse de l'utilisateur.

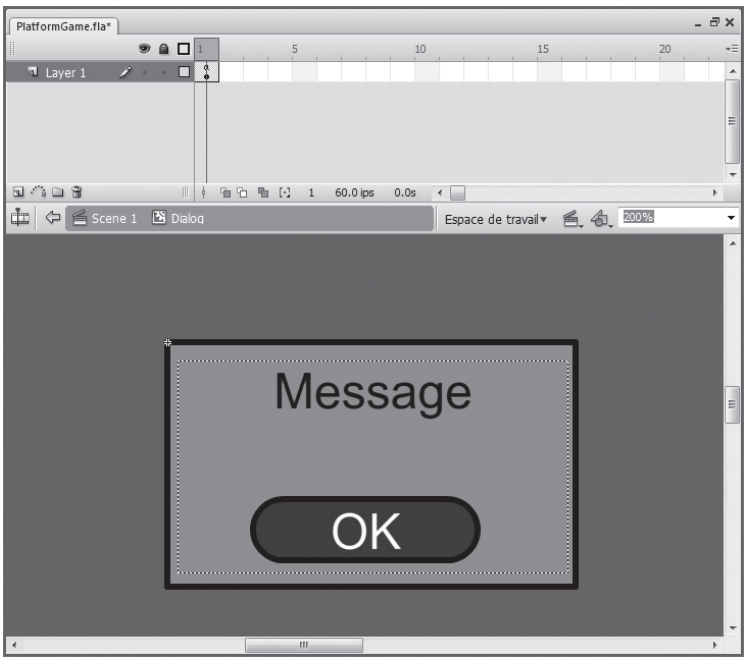


Figure 11.7
Le clip Dialog sera utilisé pour attendre que le joueur clique sur un bouton avant de continuer.

La boîte de dialogue sera affichée lorsque le joueur meurt, lorsque la partie est terminée, lorsqu'un niveau est terminé ou lorsque le joueur gagne. Lorsque l'un de ces événements se produit, le jeu s'interrompt et une boîte de dialogue s'affiche. Nous inclurons la boîte de dialogue sous forme de clip séparé avec un champ texte dynamique et un bouton.

Le scénario principal

Le scénario principal inclut une image `start` avec des instructions. Après cela, chaque image contient un clip de niveau de jeu. Ce système permet facilement de passer d'un niveau à un autre, aussi en cours de partie que lorsque vous créez les niveaux.

La seconde image contient `GameLevel1`, qui possède le nom d'instance `gamelevel`. La troisième image contient `GameLevel2`, qui possède également le nom d'instance `gamelevel`.

Lorsque le code ActionScript s'exécute, il recherche le clip avec le nom d'instance `gamelevel` dans l'image courante. Ce système permet de placer différents clips de niveaux de jeu dans différentes images.

Dans les images de niveau de jeu se trouvent trois champs texte dynamiques : un pour le niveau, un pour le nombre de vies restantes et un pour le score. La Figure 11.8 montre à quoi ressemble l'écran lorsque le jeu commence.

Figure 11.8
Trois champs texte apparaissent en bas de l'écran.

Concevoir la classe

La classe commence par examiner le clip `gamelevel`. Elle parcourt en boucle chacun des enfants de ce clip et détermine ce qu'il fait et comment il doit être représenté dans la classe du jeu.

Par exemple, si un enfant est un objet `Wall` ou `Floor`, il est ajouté à un tableau des objets de ce type. Ensuite, lorsque les personnages se déplacent, ces objets sont testés pour détecter les collisions.

Le héros et les ennemis sont aussi examinés. Le code considère que le héros possède le nom d'instance `hero` et les ennemis, les noms `enemy1`, `enemy2`, etc.

Pour déterminer le type d'objet d'un clip, nous utiliserons l'opérateur `is`*. Cet opérateur compare le type d'objet d'une variable à un type d'objet donné (par exemple* `(CetteChose is MonObjet)`*).*

La plus grande partie du code, et de loin, est celle qui gère le mouvement. Le héros peut se déplacer à gauche et à droite ou sauter. Il est cependant aussi affecté par la gravité et peut tomber d'un mur. Il peut entrer en collision avec des murs et être stoppé et entrer en collision avec le sol, ce qui interrompt sa chute.

Les ennemis en font de même, hormis que leurs mouvements ne sont pas affectés par les touches fléchées. Ils suivent néanmoins les mêmes règles que le héros.

Au lieu d'utiliser un code de mouvement différent pour le héros et les ennemis, nous leur ferons donc partager la même fonction de déplacement des personnages.

Le défilement horizontal est un autre facteur de mouvement. Le héros et les ennemis se déplaceront à l'intérieur du clip `gamelevel`. Si la position relative du héros sur la scène dépasse vers la gauche ou la droite, nous déplacerons cependant le clip `gamelevel` tout entier pour le faire défiler. Le reste du code peut ignorer cela car rien ne se déplacera véritablement à l'intérieur de `gamelevel`.

Planification des fonctions requises

Avant de commencer à programmer, voyons quelles fonctions nous allons utiliser dans la classe et lesquelles feront appel à d'autres :

- **`startPlatformGame`**. Initialise le score et les vies du joueur.
- **`startGameLevel`.** Initialise le niveau, en appelant les trois fonctions suivantes :
  - **`createHero`.** Crée l'objet `hero`, en examinant le placement de l'instance du clip `hero`.
  - **`addEnemies`**. Crée les objets `enemy`, en examinant les clips `enemyX`.
  - **`examineLevel`.** Recherche les murs, le sol et les autres éléments dans le clip `gamelevel`.
- **`keyDownFunction`.** Consigne la touche enfoncée par l'utilisateur.
- **`keyUpFunction`.** Consigne le moment où l'utilisateur a fini d'appuyer sur une touche.
- **`gameLoop`.** Appelée à chaque image pour calculer le temps écoulé et appeler les quatre fonctions suivantes :
  - **`moveEnemies`.** Parcourt en boucle tous les ennemis et les déplace.
  - **`moveCharacter`.** Déplace le personnage.
  - **`scrollWithHero`.** Fait défiler le clip `gamelevel` en fonction de l'emplacement du héros.
  - **`checkCollisions`.** Vérifie si le héros entre en collision avec des ennemis ou des objets. Appelle les trois fonctions suivantes :

  - **`enemyDie`.** Le personnage de l'ennemi est supprimé.
  - **`heroDie`.** Le héros perd une vie et la partie est terminée le cas échéant.
  - **`getObject`.** Le héros récupère un objet.
- **`addScore`.** Ajoute des points au score et l'affiche.
- **`showLives`.** Affiche le nombre de vies restantes.
- **`levelComplete`.** Le niveau est terminé, pause et affichage de la boîte de dialogue.
- **`gameComplete`.** Le trésor est trouvé, pause et affichage de la boîte de dialogue.
- **`clickDialogButton`.** L'utilisateur a cliqué sur le bouton de la boîte de dialogue, passer à l'action suivante.
- **`cleanUp`.** Supprimer la liste du jeu (`gamelist`) afin de préparer le niveau suivant.

Maintenant que nous connaissons les fonctions que nous devons écrire, créons la classe **PlatformGame.as**.

Créer la classe

Le fichier de paquetage n'est pas particulièrement long, notamment si l'on considère tout ce que ce jeu réalise. Nous conserverons donc tout le code dans une classe bien qu'il puisse être plus utile pour un jeu de plus grande envergure d'avoir des classes séparées pour les personnages, les objets et les éléments du décor.

Définition de classe

Au début de la classe figure notre liste habituelle d'instructions `import`, avec notamment la classe `flash.utils.getTimer`, dont nous avons besoin pour les animations temporelles :

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
    import flash.utils.getTimer;
```

Nous n'avons besoin que de quelques constantes. La première est `gravity` (la gravité), puis la distance jusqu'aux bords de l'écran afin de commencer le défilement horizontal.

La constante `gravity` a été déterminée suite à un processus de tentatives et d'erreurs. En sachant qu'elle serait multipliée par le nombre de millisecondes entre les étapes, j'ai commencé par une fraction réduite. Je l'ai ensuite ajustée une fois le jeu terminé, jusqu'à obtenir un comportement de saut et de chute qui m'a semblé naturel.

```
public class PlatformGame extends MovieClip {
    // Constantes de mouvement
    static const gravity:Number = .004;

    // Bord pour le défilement
    static const edgeDistance:Number = 100;
```

Lorsque le niveau de jeu est scanné, tous les objets trouvés sont placés dans deux tableaux. Le tableau `fixedObjects` contient les références à tous les objets sur lesquels le joueur peut tenir debout ou par lesquels il peut être bloqué. Le tableau `otherObjects` contient les éléments comme la clé (`Key`), la porte (`Door`), le coffre (`Chest`) et les joyaux (`Treasure`) :

```
    // Tableaux des objets
    private var fixedObjects:Array;
    private var otherObjects:Array;
```

Le clip `hero` est déjà nommé "hero" et est accessible depuis le code avec `gamelevel.hero`. L'objet `hero` dans notre classe contient cependant cette référence et bien d'autres éléments d'information concernant le personnage du héros. De la même manière, le tableau `enemies` contient une liste d'objets avec des informations concernant chaque ennemi :

```
    // Héros et ennemis
    private var hero:Object;
    private var enemies:Array;
```

Un certain nombre de variables sont requises pour tenir le registre de l'étape du jeu. Nous utiliserons `playerObjects` comme tableau pour stocker des objets que le joueur a ramassés. Le seul objet de ce type dans ce jeu est l'objet `Key`, mais nous le stockerons dans un tableau quoi qu'il en soit afin de prévoir le cas où d'autres objets pourraient être ajoutés.

`gameMode` est une chaîne qui nous aidera à indiquer aux différentes fonctions ce qui est arrivé au héros. Elle possédera d'abord la valeur `"start"` puis sera remplacée par `"play"` lorsque le jeu sera prêt à commencer.

`gameScore` et `playerLives` correspondent au nombre de points marqués et au nombre de vies restantes du joueur.

La variable `lastTime` contient la valeur en millisecondes de la dernière étape de l'animation du jeu. Nous l'utiliserons pour cadencer l'animation temporelle utilisée par les éléments du jeu :

```
    // État du jeu
    private var playerObjects:Array;
    private var gameMode:String = "start";
    private var gameScore:int;
    private var playerLives:int;
    private var lastTime:Number = 0;
```

Commencer la partie et le niveau

Lorsque le jeu commence, nous avons besoin de certaines variables d'état du jeu. Pour cela, nous appelons la fonction `startPlatformGame` avec l'image qui contient le premier niveau du jeu. Nous aurons d'autres variables qui doivent être réinitialisées à chaque niveau. Elles sont définies lorsque le `startGameLevel` est appelé à l'image suivante :

```
// Démarrer le jeu
public function startPlatformGame() {
    playerObjects = new Array();
    gameScore = 0;
    gameMode = "play";
    playerLives = 3;
}
```

La Figure 11.9 présente l'écran de démarrage du jeu, avec un bouton à cliquer pour continuer.

Figure 11.9
L'écran de démarrage du jeu de plate-forme.

La fonction *startGameLevel*

La fonction `startGameLevel` est appelée à chaque image qui contient un clip `gamelevel`. Elle délègue ensuite les tâches qui consistent à trouver et à configurer le héros, les ennemis et les objets du jeu :

```
// Niveau de départ
public function startGameLevel() {
```

```
        // Créer les personnages
        createHero();
        addEnemies();

        // Examiner le niveau et consigner tous les objets
        examineLevel();
```

La fonction `startGameLevel` configure également trois écouteurs événementiels. Le premier est la fonction principale `gameLoop`, qui s'exécutera à chaque image afin de faire avancer l'animation. Les deux autres sont les écouteurs événementiels du clavier, dont nous avons besoin pour récupérer l'entrée de l'utilisateur :

```
        // Ajouter les écouteurs
        this.addEventListener(Event.ENTER_FRAME,gameLoop);
        stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

Pour finir, le `gameMode` est positionné à "play" et deux fonctions sont appelées pour configurer l'affichage du score et des vies. L'affichage du score est mis à jour avec un appel à `addScore`, qui ajoute un certain nombre de points au score et met à jour le champ texte. Si nous ajoutons 0 point, elle agit donc comme une simple fonction d'affichage :

```
        // Configurer l'état du jeu
        gameMode = "play";
        addScore(0);
        showLives();
    }
```

La fonction *createHero*

Le clip `hero` se trouve déjà dans le clip `gamelevel`, prêt à opérer. Nous devons cependant configurer et utiliser plusieurs propriétés. Nous créerons donc un objet `hero` dans la classe pour y stocker ces propriétés :

```
    // Créer l'objet hero et définir toutes les propriétés
    public function createHero() {
        hero = new Object();
```

La première propriété est une référence au clip qui est la représentation visuelle du héros. Nous pouvons donc maintenant faire référence au héros en spécifiant `hero.mc` plutôt que `gamelevel.hero`. Cette référence fonctionnera mieux lorsque nous utilisons l'objet `hero` pour nos manipulations du personnage du joueur :

```
        hero.mc = gamelevel.hero;
```

Viennent ensuite deux propriétés qui décrivent la vélocité du héros :

```
hero.dx = 0.0;
hero.dy = 0.0;
```

La propriété `hero.inAir` se verra attribuer la valeur `true` lorsque le héros ne repose pas sur le sol ferme :

```
hero.inAir = false;
```

La propriété `hero.direction` vaudra `-1` ou `1` selon le sens dans lequel le héros est orienté :

```
hero.direction = 1;
```

La propriété `hero.animstate` contiendra la valeur `"stand"` ou la valeur `"walk"`. Avec la valeur `"walk"`, nous saurons que le personnage doit exécuter sa séquence de marche. Les images de cette séquence sont stockées dans `hero.walkAnimation`. Ici, la séquence de marche se trouve dans les images 2 à 8. Pour mémoriser l'étape qu'affiche actuellement l'animation, nous utiliserons `hero.animstep` :

```
hero.animstate = "stand";
hero.walkAnimation = new Array(2,3,4,5,6,7,8);
hero.animstep = 0;
```

La propriété `hero.jump` vaut `true` lorsque le joueur appuie sur la barre d'espace. De la même manière, `hero.moveLeft` et `hero.moveRight` prennent les valeurs `true` et `false` selon la touche fléchée qui est enfoncée :

```
hero.jump = false;
hero.moveLeft = false;
hero.moveRight = false;
```

Les quelques propriétés suivantes sont des constantes utilisées pour déterminer la hauteur du saut du personnage et la vitesse à laquelle il marche :

```
hero.jumpSpeed = .8;
hero.walkSpeed = .15;
```

Les constantes `hero.width` et `hero.height` sont utilisées pour déterminer les collisions. Au lieu d'utiliser la largeur et la hauteur effectives du personnage, qui varie selon l'image de l'animation affichée, nous utiliserons les constantes suivantes :

```
hero.width = 20.0;
hero.height = 40.0;
```

Ces constantes ont également été définies à force de tentatives et d'ajustements. J'ai commencé par des valeurs qui semblaient cohérentes, notamment en spécifiant que le personnage devait marcher environ 100 à 200 pixels par seconde. J'ai ensuite ajusté ces paramètres à mesure que le jeu se construisait.

Lorsque le héros entrera en collision avec un objet, nous le réinitialiserons à sa position de départ dans le niveau. Nous devons donc enregistrer cet emplacement pour l'utiliser à ce point :

```
        hero.startx = hero.mc.x;
        hero.starty = hero.mc.y;
    }
```

La fonction *addEnemies*

Les ennemis sont stockés dans des objets qui ressemblent simplement à l'objet `hero`. Comme les objets `hero` et `enemy` possèdent les mêmes propriétés, nous pouvons les transmettre tous deux à la fonction `moveCharacter`.

La fonction `addEnemies` recherche un clip nommé `enemy1` et l'ajoute au tableau `enemies` sous forme d'objet. Elle recherche ensuite `enemy2`, et ainsi de suite.

L'une des quelques différences entre les ennemis et le héros tient à ce que les premiers n'ont pas besoin des propriétés `startx` et `starty`. En outre, la propriété `enemy.moveRight` commence avec la valeur `true`, afin que les ennemis commencent à marcher vers la droite :

```
    // Trouver tous les ennemis dans le niveau et créer un objet pour chacun
    public function addEnemies() {
        enemies = new Array();
        var i:int = 1;
        while (true) {
            if (gamelevel["enemy"+i] == null) break;
            var enemy = new Object();
            enemy.mc = gamelevel["enemy"+i];
            enemy.dx = 0.0;
            enemy.dy = 0.0;
            enemy.inAir = false;
            enemy.direction = 1;
            enemy.animstate = "stand"
            enemy.walkAnimation = new Array(2,3,4,5);
            enemy.animstep = 0;
            enemy.jump = false;
            enemy.moveRight = true;
            enemy.moveLeft = false;
            enemy.jumpSpeed = 1.0;
            enemy.walkSpeed = .08;
            enemy.width = 30.0;
            enemy.height = 30.0;
```

```
            enemies.push(enemy);
            i++;
        }
    }
```

La fonction *examineLevel*

Une fois que le héros et tous les ennemis ont été trouvés, la fonction `examineLevel` recherche tous les enfants du clip `gamelevel` :

```
    // Rechercher tous les enfants du niveau et consigner les murs, les sols et les objets
    public function examineLevel() {
        fixedObjects = new Array();
        otherObjects = new Array();
        for(var i:int=0;i<this.gamelevel.numChildren;i++) {
            var mc = this.gamelevel.getChildAt(i);
```

Si l'objet est un objet `Floor` ou un objet `Wall`, il est ajouté au tableau `fixedObjects` sous forme d'objet avec une référence au clip, mais également d'autres informations. Les emplacements des quatre côtés sont stockés dans `leftside`, `rightside`, `topside` et `bottomside`. Nous avons besoin de pouvoir y accéder rapidement au moment de déterminer les collisions :

```
            // Ajouter les sols et les murs à fixedObjects
            if ((mc is Floor) || (mc is Wall)) {
                var floorObject:Object = new Object();
                floorObject.mc = mc;
                floorObject.leftside = mc.x;
                floorObject.rightside = mc.x+mc.width;
                floorObject.topside = mc.y;
                floorObject.bottomside = mc.y+mc.height;
                fixedObjects.push(floorObject);
    All other objects are added very simply to the otherObjects array:
            // Ajouter les joyaux, le coffre, la clé et la porte à otherOjects
            } else if ((mc is Treasure) || (mc is Key) ||
    (mc is Door) || (mc is Chest)) {
                otherObjects.push(mc);
            }
        }
    }
```

Entrée clavier

L'acceptation de l'entrée clavier fonctionnera comme dans les précédents jeux, en utilisant les touches fléchées. Nous positionnerons cependant directement les propriétés `moveLeft`, `moveRight` et `jump` du `hero`. Nous n'autoriserons en outre `jump` à prendre la valeur `true` que si le `hero` ne se trouve pas déjà dans les airs :

```
// Consigner appuis sur les touches, définir propriétés de hero
public function keyDownFunction(event:KeyboardEvent) {
    if (gameMode != "play") return; // Ne pas bouger avant le mode jeu

    if (event.keyCode == 37) {
        hero.moveLeft = true;
    } else if (event.keyCode == 39) {
        hero.moveRight = true;
    } else if (event.keyCode == 32) {
        if (!hero.inAir) {
            hero.jump = true;
        }
    }
}
```

La fonction `keyUpFunction` reconnaît le moment où le joueur relâche la touche et désactive alors les drapeaux `moveLeft` et `moveRight` :

```
public function keyUpFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        hero.moveLeft = false;
    } else if (event.keyCode == 39) {
        hero.moveRight = false;
    }
}
```

La boucle principale du jeu

Grâce à l'écouteur `EVENT_FRAME`, la fonction `gameLoop` sera appelée une fois par image. Elle détermine le nombre de millisecondes qui se sont écoulées depuis la dernière fois où elle a été appelée.

Si `gameMode` vaut `"play"`, elle appelle une variété de fonctions. Pour commencer, elle appelle `moveCharacter` avec en paramètre le `hero` comme objet, mais également `timeDiff`.

Ensuite, elle appelle `moveEnemies`, qui s'occupe de parcourir en boucle tous les ennemis et appelle `moveCharacter` pour chacun d'entre eux.

La fonction `checkForCollisions` verra si des ennemis entrent en collision avec le héros ou si le héros touche un objet.

Pour finir, `scrollWithHero` synchronisera, si nécessaire, le clip `gamelevel` avec la position du héros :

```
public function gameLoop(event:Event) {

    // Calculer le temps écoulé
    if (lastTime == 0) lastTime = getTimer();
    var timeDiff:int = getTimer()-lastTime;
    lastTime += timeDiff;

    // Ne réaliser des tâches qu'en mode Jeu
    if (gameMode == "play") {
        moveCharacter(hero,timeDiff);
        moveEnemies(timeDiff);
        checkCollisions();
        scrollWithHero();
    }
}
```

La fonction `moveEnemies` vérifie les propriétés `hitWallRight` et `hitWallLeft` de chaque ennemi. Il s'agit de propriétés spéciales attribuées à l'objet du personnage lorsque celui-ci est traité par `moveCharacter`. Nous les utilisons non pas pour l'objet `hero`, mais pour les ennemis.

Lorsqu'un ennemi touche un mur, nous inversons son sens de marche :

```
public function moveEnemies(timeDiff:int) {
    for(var i:int=0;i<enemies.length;i++) {

        // Bouger
        moveCharacter(enemies[i],timeDiff);

        // Si l'ennemi rencontre un mur, changer de sens
        if (enemies[i].hitWallRight) {
            enemies[i].moveLeft = true;
            enemies[i].moveRight = false;
        } else if (enemies[i].hitWallLeft) {
            enemies[i].moveLeft = false;
            enemies[i].moveRight = true;
        }
    }
}
```

Il pourrait être souhaitable que différents ennemis possèdent différents comportements. Par exemple, vous pourriez vérifier si le héros se trouve à gauche ou à droite de l'ennemi et ne vous déplacer que dans ce sens. Vous pourriez sinon vérifier la distance entre le héros et l'ennemi et ne bouger que si le héros est proche.

Mouvement des personnages

Il est maintenant temps d'examiner le cœur du jeu : la fonction `moveCharacter`. Elle prend en paramètre un objet de personnage, qu'il s'agisse du `hero` ou d'un `enemy`, et le stocke dans `char`. Elle prend aussi en paramètre le nombre de millisecondes qui se sont écoulées et le stocke dans `timeDiff` :

```
public function moveCharacter(char:Object,timeDiff:Number) {
```

Au début du jeu, la variable `lastTime` est initialisée et le temps écoulé vaut 0. La valeur 0 ne produirait pas un résultat satisfaisant avec certains des calculs de vélocité, aussi nous interrompons la fonction à ce point si `timeDiff` vaut `0` :

```
if (timeDiff < 1) return;
```

Si `timeDiff` vaut `0`, `verticalChange` vaudra `0` également. Si `verticalChange` vaut `0`, la nouvelle position verticale et l'ancienne position verticale seront les mêmes, ce qui permet difficilement de savoir si le personnage repose sur le sol ou flotte dans les airs.

La première chose que nous devons faire est de calculer le changement vertical dû à la gravité. La gravité agit toujours sur nous, même lorsque nous nous tenons debout sur le sol. Nous calculerons le changement de vélocité et de position verticale du personnage en fonction de la constante `gravity` et du temps qui s'est écoulé.

Pour déterminer le changement sur l'axe vertical en raison de la gravité dans l'animation temporelle, nous prenons la vitesse verticale actuelle (`char.dy`) et la multiplions par `timeDiff`. Ce calcul concerne la vitesse vers le haut ou vers le bas du personnage.

Ensuite, nous ajoutons `timeDiff` multiplié par `gravity` pour déterminer la distance parcourue depuis que la dernière vitesse verticale (`dy`) a été mise à jour.

Ensuite, nous modifions la vitesse verticale pour la suite en ajoutant `gravity*timeDiff` :

```
// Le personnage est tiré vers le bas par la gravité
var verticalChange:Number = char.dy*timeDiff + timeDiff*gravity;
if (verticalChange > 15.0) verticalChange = 15.0;
char.dy += timeDiff*gravity;
```

Vous remarquerez que `verticalChange` est limitée à `15.0`. Il s'agit de ce que l'on appelle la vélocité terminale. Dans la vie, cela se produit lorsque la résistance au vent contrecarre l'accélération due à la gravité et que l'objet ne peut plus tomber plus vite. Nous ajoutons cette limite ici parce que, si le personnage tombe d'une grande distance, il accélérera sa chute assez sensiblement et l'effet visuel ne paraîtra pas naturel.

Avant d'examiner les mouvements vers la gauche et la droite, nous allons opérer quelques présuppositions concernant ce qui va se passer. Nous supposerons que l'état de l'animation sera `"stand"` et que le nouveau sens de progression du personnage sera le même que le sens actuel. Nous présupposerons également qu'il n'y aura pas de changement horizontal de position :

```
// Réagir aux changements d'appui sur les touches
var horizontalChange = 0;
var newAnimState:String = "stand";
var newDirection:int = char.direction;
```

Ensuite, nous vérifierons immédiatement le bien-fondé de cette présupposition en examinant les propriétés `char.moveLeft` et `char.moveRight`. Elles auront été définies dans la fonction `keyDownFunction` si le joueur enfonce la touche fléchée de gauche ou de droite.

Si la touche de gauche est enfoncée, `horizontalChange` se voit attribuer la valeur `char.walkSpeed *timeDiff` en négatif. En outre, `newDirection` prend la valeur `-1`. Si la touche de droite est enfoncée, `horizontalChange` se voit attribuer la valeur `char.walkSpeed*timeDiff` positive et `newDirection` prend la valeur `1`. Dans les deux cas, `newAnimState` récupère la valeur `"walk"` :

```
if (char.moveLeft) {
    // Marche vers la gauche
    horizontalChange = -char.walkSpeed*timeDiff;
    newAnimState = "walk";
    newDirection = -1;
} else if (char.moveRight) {
    // Marche vers la droite
    horizontalChange = char.walkSpeed*timeDiff;
    newAnimState = "walk";
    newDirection = 1;
}
```

La prochaine vérification que nous allons opérer concerne `char.jump`, qui se verra attribuer la valeur `true` lorsque le joueur appuie sur la barre d'espace. Nous réattribuerons immédiatement la valeur `false` afin que cette action ne se produise qu'une seule fois par appui sur la barre d'espace.

Ensuite, nous changerons `char.dy` en lui attribuant la valeur en négatif de la constante `char.jumpSpeed`. Ce réglage donnera au personnage un élan vers le haut correspondant à la force initiale du saut.

Nous donnerons aussi à `newAnimState` la valeur `"jump"`. La Figure 11.10 montre l'état de saut du héros.

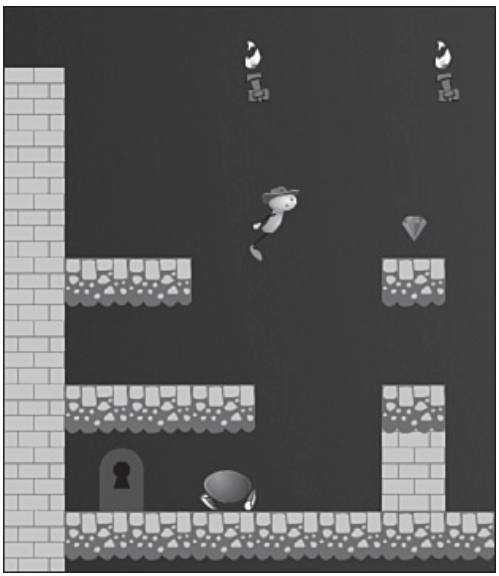


Figure 11.10
Le héros prend cette apparence à chaque fois qu'il est en l'air.

```
if (char.jump) {
    // Début du saut
    char.jump = false;
    char.dy = -char.jumpSpeed;
    verticalChange = -char.jumpSpeed;
    newAnimState = "jump";
}
```

Nous sommes maintenant sur le point d'examiner les `fixedObjects` dans la scène afin de vérifier les collisions de mouvements. Avant cela, nous présupposerons qu'il n'y a pas de collision avec un mur de gauche ou de droite et que le personnage reste en l'air :

```
// Présupposition : pas de choc contre un mur et suspension en l'air
char.hitWallRight = false;
char.hitWallLeft = false;
char.inAir = true;
```

Nous calculerons la nouvelle position verticale du personnage en fonction de la position actuelle et de la variable `verticalChange` définie précédemment :

```
// Calculer la nouvelle position verticale
var newY:Number = char.mc.y + verticalChange;
```

Nous allons maintenant examiner chaque objet fixe et voir si l'un d'entre eux se trouve sous les pieds du personnage. Pour cela, nous commençons par voir si le personnage est aligné horizontalement avec l'objet. S'il est trop loin à gauche ou à droite, nous pourrons cesser immédiatement d'examiner l'objet.

La Figure 11.11 présente une illustration du problème. Le rectangle A présente le personnage dans la position actuelle et le rectangle B, dans sa prochaine position. Vous pouvez remarquer que le bas du personnage se trouve juste au-dessus du sol dans A et en dessous dans B.

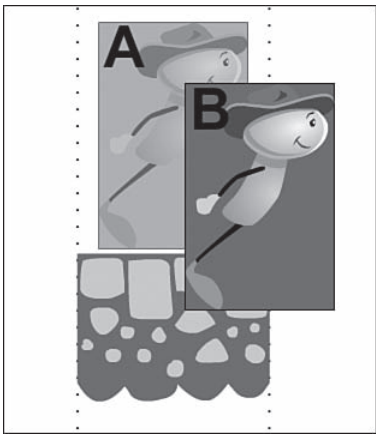


Figure 11.11
En une étape, le personnage passerait au travers du sol si notre code ne l'arrêtait pas.

Nous verrons ensuite si le personnage est actuellement au-dessus de l'objet et si son emplacement `newY` est en dessous. Cela signifie que le personnage passerait normalement au travers de l'objet. Rappelez-vous que le point d'alignement des personnages se trouve à l'extrémité inférieure de leurs pieds et que celui des murs et des sols se trouve à leur extrémité supérieure.

Au lieu de laisser le personnage traverser l'objet, nous allons le stopper directement à la surface supérieure de l'objet. La propriété `char.dy` sera positionnée à `0` et la propriété `char.inAir`, à `false` :

```
// Parcourir en boucle tous les objets fixes et voir si le personnage a atterri
for(var i:int=0;i<fixedObjects.length;i++) {
    if ((char.mc.x+char.width/2 > fixedObjects[i].leftside) && (char.mc.x-char.width/2
     < fixedObjects[i].rightside)) {
        if ((char.mc.y <= fixedObjects[i].topside) && (newY > fixedObjects[i].topside)) {
            newY = fixedObjects[i].topside;
            char.dy = 0;
            char.inAir = false;
            break;
        }
    }
}
```

Pendant qu'un personnage repose sur le dessus d'un objet `Floor` ou `Wall`, ce test vertical est réalisé à chaque étape et à chaque étape il aura pour résultat de conserver le personnage au-dessus du sol.

Nous réaliserons ensuite un test similaire avec la position horizontale. Nous créerons une variable `newX` avec le nouvel emplacement horizontal, en supposant qu'il n'y a pas de collision :

```
// Trouver nouvelle position horizontale
var newX:Number = char.mc.x + horizontalChange;
```

Nous examinerons maintenant chacun des objets `Wall` et `Floor` et verrons si aucun ne correspond verticalement. Si c'est le cas, nous verrons s'ils sont traversés avec la transition allant de la position courante à la nouvelle.

Nous devons vérifier le côté gauche et le côté droit. Si l'un des tests vaut `true`, la position `x` est définie de manière à correspondre exactement au mur et `char.hitWallLeft` ou `char.hitWallRight` prend la valeur `true` :

```
// Parcourir en boucle tous les objets pour voir si en ligne personnage est rentré
   dans un mur
for(i=0;i<fixedObjects.length;i++) {
    if ((newY > fixedObjects[i].topside) && (newY-char.height < fixedObjects[i].bottomside)) {
        if ((char.mc.x-char.width/2 >= fixedObjects[i].rightside) && (newX-char.width/2
        <= fixedObjects[i].rightside)) {
            newX = fixedObjects[i].rightside+char.width/2;
            char.hitWallLeft = true;
            break;
    }
        if ((char.mc.x+char.width/2 <= fixedObjects[i].leftside) &&
            (newX+char.width/2 >= fixedObjects[i].leftside)) {
            newX = fixedObjects[i].leftside-char.width/2;
            char.hitWallRight = true;
            break;
        }
    }
}
```

Nous connaissons maintenant la nouvelle position du personnage, en tenant compte de la vitesse horizontale et verticale, de la gravité et des collisions avec le sol et les murs. Nous pouvons définir l'emplacement du personnage :

```
// Définir l'emplacement du personnage
char.mc.x = newX;
char.mc.y = newY;
```

Le reste de la fonction s'occupe de l'apparence du personnage. Nous vérifions la valeur `char.inAir` ; si elle vaut `true` à ce stade, nous devons donner à `newAnimState` la valeur `"jump"` :

```
// Définir l'état de l'animation
if (char.inAir) {
    newAnimState = "jump";
}
```

Nous avons donc fini de changer `newAnimState`. Cette variable a démarré avec la valeur `"stand"`. Ensuite, elle est passée à `"walk"` si la touche fléchée de gauche ou de droite a été enfoncée. Elle a pu aussi passer à `"jump"` si le joueur a appuyé sur la barre d'espace ou si le personnage se trouvait dans les airs. Nous allons maintenant définir l'état de l'animation (`animstate`) en lui attribuant la valeur de `newAnimState` :

```
char.animstate = newAnimState;
```

Ensuite, nous utiliserons `animstate` pour décider de l'apparence à donner au personnage. Si celui-ci marche, `animstep` est augmentée d'une fraction de `timeDiff` et une vérification est opérée pour voir si `animstep` doit reboucler vers 0. Ensuite, l'image du personnage est définie en fonction de celle spécifiée dans `walkAnimation` :

```
    // Avancer dans le cycle de marche
    if (char.animstate == "walk") {
        char.animstep += timeDiff/60;
        if (char.animstep > char.walkAnimation.length) {
            char.animstep = 0;
        }
        char.mc.gotoAndStop(char.walkAnimation[Math.floor(char.animstep)]);
If the character is not walking, we'll set the frame to either "stand" or "jump"
depending on the value of animstate:
    // Pas de marche, afficher l'état debout ou saut
    } else {
        char.mc.gotoAndStop(char.animstate);
    }
```

La dernière tâche dont `moveCharacter` doit se charger est de définir l'orientation du personnage. La propriété `direction` vaut `-1` pour s'orienter vers la gauche et `1` pour s'orienter vers la droite. Nous la remplirons avec la valeur de `newDirection` qui a été déterminée précédemment. Ensuite, nous définirons la propriété `scaleX` du personnage.

Le positionnement de la propriété `scaleX` d'un sprite ou d'un clip constitue un moyen simple de renverser n'importe quel objet. Toutefois, si vous avez des ombres ou des effets 3D dans les graphismes de l'objet, vous devez dessiner une autre version de l'objet qui pointe dans la direction opposée ; sans cela, le personnage inversé ne présenterait pas l'apparence souhaitée.

```
        // Directions modifiées
        if (newDirection != char.direction) {
            char.direction = newDirection;
            char.mc.scaleX = char.direction;
        }
    }
```

Défilement du niveau du jeu

`scrollWithHero` est une autre fonction qui s'exécute à chaque image. Elle vérifie la position du héros par rapport à la scène. `stagePosition` est calculée en ajoutant `gamelevel.x` à `hero.mc.x`. Ensuite, nous obtenons également `rightEdge` et `leftEdge` en fonction des bords de l'écran, moins la constante `edgeDistance`. C'est à ces points que l'écran commence à défiler si besoin.

Si le héros dépasse le bord droit (`rightEdge`), la position du niveau (`gamelevel`) est déplacée de la même distance vers la gauche. Si `gamelevel` arrive trop loin à gauche, il est cependant empêché d'avancer afin que l'extrémité droite de `gamelevel` se trouve du côté droit de la scène.

De la même manière, si le héros est assez loin à gauche, le clip `gamelevel` avance à droite, mais au maximum jusqu'au moment où le côté de `gamelevel` passerait à droite du côté gauche de l'écran :

```
    // Faire défiler vers la droite ou la gauche en cas de besoin
    public function scrollWithHero() {
        var stagePosition:Number = gamelevel.x+hero.mc.x;
        var rightEdge:Number = stage.stageWidth-edgeDistance;
        var leftEdge:Number = edgeDistance;
        if (stagePosition > rightEdge) {
            gamelevel.x -= (stagePosition-rightEdge);
            if (gamelevel.x < -(gamelevel.width-stage.stageWidth))
    gamelevel.x = -(gamelevel.width-stage.stageWidth);
        }
        if (stagePosition < leftEdge) {
            gamelevel.x += (leftEdge-stagePosition);
            if (gamelevel.x > 0) gamelevel.x = 0;
        }
    }
```

Vérifier les collisions

La fonction `checkCollisions` parcourt en boucle tous les ennemis et tous les autres objets (`otherObjects`). Elle utilise une fonction `hitTestObject` simple pour chaque objet afin de réaliser les tests de collision.

Si la collision du `hero` et de l'`enemy` intervient pendant que le `hero` se trouve en l'air et progresse vers le bas, l'ennemi est détruit en appelant `enemyDie`. Si ce n'est pas le cas, nous appelons `heroDie` à la place :

```
// Vérifier les collisions avec les ennemis et les objets
public function checkCollisions() {

    // Ennemis
    for(var i:int=enemies.length-1;i>=0;i--) {
        if (hero.mc.hitTestObject(enemies[i].mc)) {

            // Le héros saute-t-il vers le bas sur l'ennemi ?
            if (hero.inAir && (hero.dy > 0)) {
                enemyDie(i);
            } else {
                heroDie();
            }
        }
    }
```

Si le `hero` entre en collision avec un objet dans la liste `otherObjects`, nous appelons `getObject` en lui passant le numéro de l'élément dans la liste :

```
    // Objets
    for(i=otherObjects.length-1;i>=0;i--) {
        if (hero.mc.hitTestObject(otherObjects[i])) {
            getObject(i);
        }
    }
}
```

Mort des ennemis et du joueur

Lorsqu'un objet `enemy` est détruit, il est supprimé du clip `gamelevel` et de la liste `enemies`. Il n'y a besoin de rien d'autre pour le faire disparaître.

Nous allons tout de même insérer un effet spécial. En utilisant la classe `PointBurst` du Chapitre 8, nous ferons apparaître un petit message à l'endroit où l'ennemi a été supprimé, en affichant les mots "Got Em!" (Touchés !). La Figure 11.12 présente l'écran juste après que l'ennemi a été détruit.

```
// Supprimer l'ennemi
public function enemyDie(enemyNum:int) {
    var pb:PointBurst = new PointBurst(gamelevel,
"Got Em!",enemies[enemyNum].mc.x,
enemies[enemyNum].mc.y-20);
    gamelevel.removeChild(enemies[enemyNum].mc);
    enemies.splice(enemyNum,1);
}
```

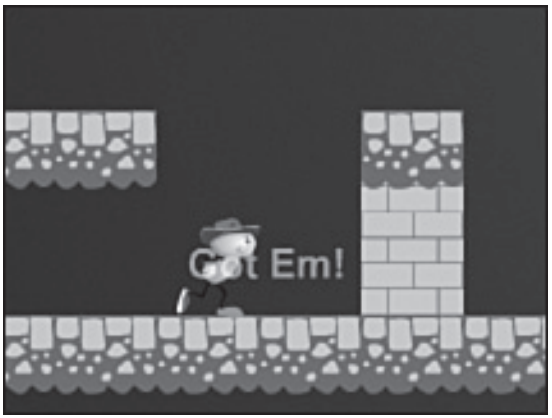


Figure 11.12
Les mots "Got Em!" apparaissent à l'endroit où se trouvait l'ennemi, changent de taille et disparaissent rapidement.

*Pour utiliser la classe `PointBurst`, vous devez en faire glisser une copie dans le même dossier que les fichiers **PlatformGame.fla** et **PlatformGame.as**. Vous aurez aussi besoin d'ajouter la police Arial à la bibliothèque de **PlatformGame.fla** et de la configurer de manière à l'exporter pour ActionScript.*

Lorsque le joueur meurt après être entré dans un ennemi, nous avons l'opportunité d'afficher la boîte de dialogue conçue précédemment.

Pour créer la boîte de dialogue, il nous suffit de créer un nouvel objet `Dialog` et de l'attribuer à une variable temporaire. Ensuite, nous définissons les positions `x` et `y` et utilisons `addChild` pour le placer dans la scène.

Nous vérifions après cela le nombre de vies du joueur (`playerLives`). S'il est nul, nous affichons le texte "Game Over!" (partie terminée) dans la boîte de dialogue et donnons à `gameMode` la valeur `"gameover"`. S'il reste des vies, nous soustrayons une unité au compte, affichons le message "He Got You!" (il vous a eu) et donnons à `gameMode` la valeur `"dead"`.

La variable `gameMode` joue un rôle important quant à ce qui se passe lorsque le joueur clique sur le bouton dans la boîte de dialogue :

```
// L'ennemi a eu le joueur
public function heroDie() {
    // Afficher la boîte de dialogue
    var dialog:Dialog = new Dialog();
    dialog.x = 175;
    dialog.y = 100;
    addChild(dialog);
```

```
        if (playerLives == 0) {
            gameMode = „gameover";
            dialog.message.text = „Game Over!";
        } else {
            gameMode = „dead";
            dialog.message.text = „He Got You!";
            playerLives--;
        }

        hero.mc.gotoAndPlay(„die");
    }
```

La dernière tâche dont s'occupe la fonction `heroDie` consiste à demander au clip `hero` de s'exécuter à partir de l'image `die`. Cette fonction commence une animation qui montre le personnage qui tombe en arrière. Une commande `stop` figure à la fin du scénario de `hero` afin que le clip ne boucle pas pour revenir au début. La Figure 11.13 montre le héros mort et la boîte de dialogue.

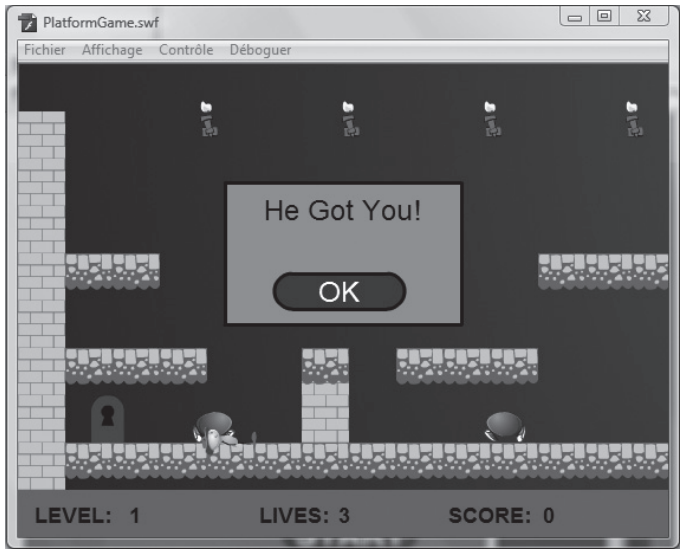


Figure 11.13
Le héros est mort et le joueur doit maintenant cliquer sur le bouton pour commencer une nouvelle vie.

Collecter des points et des objets

Lorsque le joueur entre en collision avec un objet du tableau `otherObjects`, il obtient des points, récupère un objet dans son inventaire ou termine le niveau.

Si le type de l'objet est `Treasure`, le joueur récupère 100 points. Nous utiliserons la classe `PointBurst` à nouveau pour afficher le texte 100 à l'emplacement concerné. Ensuite, nous supprimerons l'objet de `gamelevel` et de `otherObjects`. Nous appellerons la fonction `addScore` pour ajouter 100 points et mettrons à jour le score :

```
// Le joueur entre en collision avec des objets
public function getObject(objectNum:int) {

    // Attribuer des points pour les joyaux
    if (otherObjects[objectNum] is Treasure) {
        var pb:PointBurst = new PointBurst(gamelevel,100,
otherObjects[objectNum].x,otherObjects[objectNum].y);
        gamelevel.removeChild(otherObjects[objectNum]);
        otherObjects.splice(objectNum,1);
        addScore(100);
```

L'une des solutions simples pour avoir différentes valeurs de point pour les différents joyaux consiste à utiliser le nom d'instance de l'objet `Treasure`. En l'état du jeu, ce nom n'est utilisé par rien d'autre. Vous pourriez donc attribuer le nom "100" à un objet `Treasure` et "200" à un autre. Ensuite, il suffirait d'examiner `otherObjects[numObjet].name` pour attribuer le nombre de points correspondant.

Si l'objet est une clé (`Key`), nous utilisons `PointBurst` pour afficher le message "Got Key!" (Clé récupérée !). Nous ajoutons la chaîne `"Key"` au tableau `playerObjects` qui sert d'inventaire. L'objet est ensuite supprimé de `gamelevel` et `otherObjects` :

```
// Clé récupérée, ajouter à l'inventaire
} else if (otherObjects[objectNum] is Key) {
    pb = new PointBurst(gamelevel,"Got Key!",otherObjects[objectNum].x,otherObjects
        [objectNum].y);
    playerObjects.push("Key");
    gamelevel.removeChild(otherObjects[objectNum]);
    otherObjects.splice(objectNum,1);
```

L'objet peut aussi être une porte (`Door`). Dans ce cas, nous examinons l'inventaire `playerObjects` afin de vérifier si l'élément `"Key"` y figure. Si le joueur a eu la clé, la porte s'ouvre. Nous ferons cela en demandant au clip `Door` de lancer sa lecture à partir de l'image `open`. Ensuite, nous appellerons `levelComplete`, qui affiche une boîte de dialogue :

```
// Porte atteinte, terminer le niveau si le héros a la clé
} else if (otherObjects[objectNum] is Door) {
    if (playerObjects.indexOf("Key") == -1) return;
    if (otherObjects[objectNum].currentFrame == 1) {
```

```
                otherObjects[objectNum].gotoAndPlay("open");
                levelComplete();
            }
```

La dernière possibilité est que le joueur ait trouvé le coffre (`Chest`). Ce cas signale la fin du second niveau et la fin de la quête du joueur. Ce clip possède également une image `open`, mais nous utiliserons `gotoAndStop` car il n'y a pas d'animation à cet endroit. Ensuite, nous appelons `gameComplete` :

```
        // Coffre trouvé, jeu gagné
        } else if (otherObjects[objectNum] is Chest) {
            otherObjects[objectNum].gotoAndStop("open");
            gameComplete();
        }
    }
```

Afficher l'état du joueur

Il est maintenant temps de considérer certaines fonctions utilitaires. Ces fonctions sont appelées à différents endroits dans le jeu en cas de besoin. La première ajoute un certain nombre de points à la variable `gameScore`, puis met à jour le champ texte `scoreDisplay` dans la scène :

```
// Ajouter des points au score
public function addScore(numPoints:int) {
    gameScore += numPoints;
    scoreDisplay.text = String(gameScore);
}
```

La fonction qui suit place la valeur de `playerLives` dans le champ texte `livesDisplay` :

```
// Mettre à jour les vies du joueur
public function showLives() {
    livesDisplay.text = String(playerLives);
}
```

Terminer les niveaux et le jeu

Le premier niveau se termine lorsque le joueur récupère la clé et ouvre la porte. Le second se termine lorsque le joueur trouve le coffre du trésor. Dans les deux cas, un objet `Dialog` est créé et placé au centre de l'écran.

Dans le cas où la porte est ouverte et le niveau un, terminé, la boîte de dialogue affiche le message "Level Complete!" (niveau terminé) et `gameMode` se voit attribuer la valeur `"done"` :

```
// Niveau terminé, afficher la boîte de dialogue
public function levelComplete() {
    gameMode = "done";
    var dialog:Dialog = new Dialog();
```

```
            dialog.x = 175;
            dialog.y = 100;
            addChild(dialog);
            dialog.message.text = „Level Complete!";
        }
```

Si le deuxième niveau se termine après que le joueur a trouvé le coffre, le message `"You Got the Treasure!"` apparaît et `gameMode` récupère la valeur `"gameover"` :

```
        // Partie terminée, afficher la boîte de dialogue
        public function gameComplete() {
            gameMode = "gameover";
            var dialog:Dialog = new Dialog();
            dialog.x = 175;
            dialog.y = 100;
            addChild(dialog);
            dialog.message.text = „You Got the Treasure!";
        }
```

La boîte de dialogue du jeu

La boîte de dialogue apparaît lorsque le joueur est mort, a terminé un niveau ou a fini le jeu. Lorsque le joueur clique sur le bouton de cette boîte de dialogue, nous appelons la fonction `clickDialogButton` dans la classe principale. Voici le code qui se trouve dans l'objet `Dialog` :

```
        okButton.addEventListener(MouseEvent.CLICK,MovieClip(parent).clickDialogButton);
```

La première tâche dont se charge la fonction `clickDialogButton` consiste à supprimer la boîte de dialogue elle-même :

```
        // Le joueur a cliqué sur le bouton de la boîte de dialogue
        public function clickDialogButton(event:MouseEvent) {
            removeChild(MovieClip(event.currentTarget.parent));
```

Sa prochaine action dépend de la valeur de `gameMode`. Si le joueur est mort, l'affichage du nombre de vies est mis à jour, le héros est replacé au début du niveau et `gameMode` récupère de nouveau la valeur `"play"` afin que le jeu se poursuive :

```
            // Nouvelle vie, recommencer ou passer au niveau suivant
            if (gameMode == "dead") {
                // Réinitialisation du héros
                showLives();
                hero.mc.x = hero.startx;
                hero.mc.y = hero.starty;
                gameMode = "play";
```

Si `gameMode` vaut `"gameover"` (ce qui se produit lorsque le joueur meurt pour la dernière fois ou lorsque le joueur trouve le coffre), nous appelons la fonction `cleanUp` pour supprimer le clip `gamelevel` et nous ramenons l'animation au début :

```
} else if (gameMode == "gameover") {
    cleanUp();
    gotoAndStop("start");
```

L'autre possibilité concerne le cas où `gameMode` vaut `"done"`. Cela signifie qu'il est temps de passer au niveau suivant. La fonction `cleanUp` est appelée de nouveau et l'animation est envoyée à l'image suivante, où une nouvelle version de `gamelevel` attend :

```
} else if (gameMode == "done") {
    cleanUp();
    nextFrame();
}
```

L'une des dernières choses à effectuer consiste à ramener le focus du clavier sur la scène. La scène perd le focus lorsque l'utilisateur clique sur le bouton. Nous souhaitons nous assurer que les touches fléchées et la barre d'espace seront de nouveau acheminées vers la scène :

```
    // Redonner à la scène le focus clavier
    stage.focus = stage;
}
```

La fonction `cleanUp`, qu'appelle la fonction `clickDialogButton`, supprime le `gamelevel` ainsi que les écouteurs qui étaient appliqués à la scène et l'écouteur `ENTER_FRAME`. Ces écouteurs seront recréés dans `startLevel` si le joueur doit passer au niveau suivant :

```
// Nettoyer le jeu
public function cleanUp() {
    removeChild(gamelevel);
    this.removeEventListener(Event.ENTER_FRAME,gameLoop);
    stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
    stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
}
```

Modifier le jeu

Pour que ce jeu devienne vraiment palpitant, il conviendrait d'y ajouter d'autres niveaux contenant plus d'éléments. Vous pouvez ajouter autant de niveaux que vous le souhaitez.

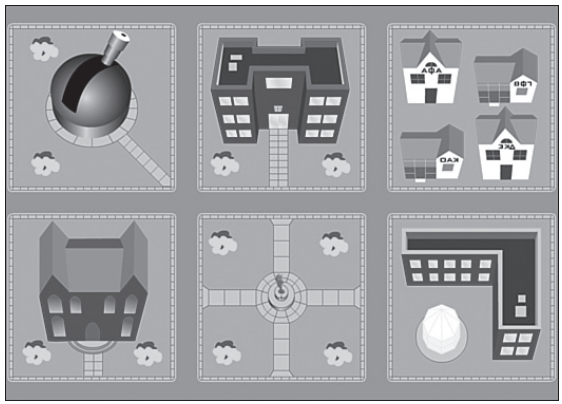
Pour l'instant, le premier niveau se termine lorsque la clé est dans l'inventaire et que la porte est trouvée. Il pourrait être intéressant de modifier le code afin d'ajouter d'autres options, par exemple avec une porte qui ne requiert pas de clé ou une porte qui en nécessite plusieurs.

L'autre fonctionnalité qui pourrait rendre le jeu plus attrayant consisterait à avoir plus de moyens de mourir. Pour l'instant, il n'est possible de mourir que si un ennemi vous touche alors que vous ne sautez pas et il est assez facile de se débarrasser des ennemis.

Que se passerait-il en revanche si des pierres ou des objets pouvaient vous tuer également ? Par exemple, la scène pourrait contenir des fosses de lave animée qu'il faudrait impérativement sauter sans les toucher pour parvenir au but.

Il pourrait aussi y avoir d'autres obstacles animés, comme des flèches qui jaillissent des murs. Ces objets se comporteraient comme des ennemis, mais "mourraient" en atteignant le mur opposé. Vous pourriez aussi définir un `Timer` pour créer une nouvelle flèche à intervalles réguliers.

Les possibilités sont quasiment infinies. Pourtant, s'il est aisé de créer un jeu de plate-forme créatif et amusant, il est tout aussi facile d'en créer un mauvais. Concevez donc soigneusement votre jeu et testez et ajustez-le au fur et à mesure.



12

Jeux de mondes : jeux de conduite et d'exploration

Au sommaire de ce chapitre

- Créer un jeu de conduite en vue aérienne
- Créer un jeu de course

Au chapitre précédent, vous avez vu comment il était possible de créer un petit monde à l'intérieur d'un jeu ActionScript. Ce type de jeu de plate-forme crée une vue latérale qui est souvent utilisée pour les jeux d'aventure en intérieur et les quêtes.

Un autre type de jeu de monde peut être créé avec une vue aérienne. Le joueur se déplace alors dans une carte. Ce type de représentation peut convenir à n'importe quel genre de scénario et de thème. J'ai cependant récemment remarqué qu'il existait un grand nombre de jeux de ce genre où le joueur conduisait un véhicule dans une ville ou une autre sorte de site extérieur.

Dans ce chapitre, nous allons créer un jeu de conduite en vue aérienne et un jeu de course simple. Ces deux types de jeux ont un certain nombre de points communs.

Créer un jeu de conduite en vue aérienne

Créons un jeu de conduite simple en vue aérienne. Ce jeu contient une carte détaillée, une voiture, des objets à ramasser et une logique de jeu complexe qui implique un emplacement où déposer les objets collectés.

Codes sources

http://flashgameu.com

A3GPU12_TopDownGame.zip

Créer un jeu en vue aérienne

Notre jeu d'exemple pour ce chapitre met en scène un campus de lycée. Une zone de trois blocs sur trois inclut différents immeubles qui remplissent l'espace entre les rues (voir Figure 12.1).

Si vous examinez attentivement la grille en bas de la carte, vous verrez une petite voiture. Il s'agit de la voiture du joueur, qu'il peut "conduire" dans la carte.

Vu les dimensions de la carte, le joueur ne pourra pas en voir plus d'une petite section à la fois. La carte fait 2 400 pixels carrés et l'écran n'en fait que 550 × 400.

À mesure que le joueur conduit, la carte se repositionne en conservant la voiture exactement au centre de la scène.

La Figure 12.2 présente l'écran lorsque le joueur commence. Vous pouvez voir la grille en bas et une portion du parking au-dessus. Une bande semi-transparente avec les éléments du score apparaît en bas.

La carte est logée dans sa totalité dans un unique clip nommé `GameMap`. À l'intérieur, les neuf groupes d'immeubles possèdent chacun leur propre clip pour des raisons pratiques d'organisation. Les rues sont composées de morceaux droits et de trois différents types de coins. La clôture externe est constituée de différentes pièces également.

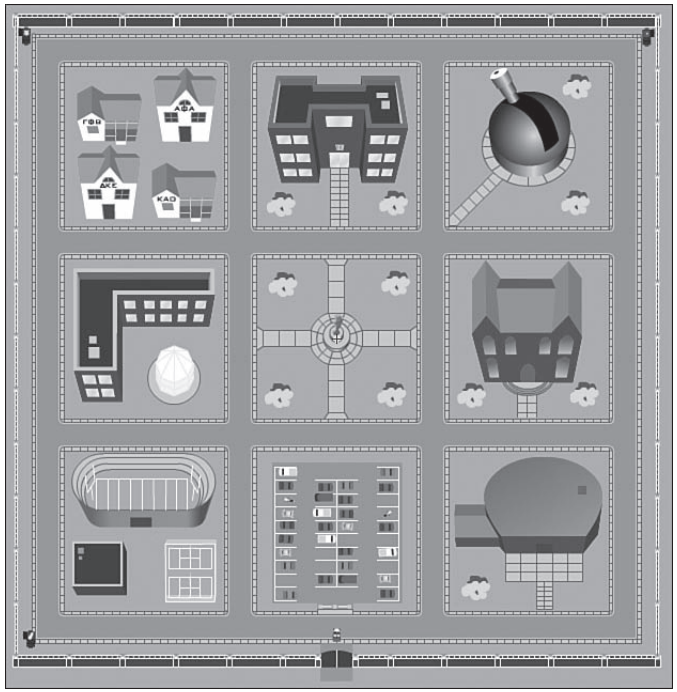


Figure 12.1
L'univers entier du jeu fait environ 2 400 pixels de large et 2 400 de haut.

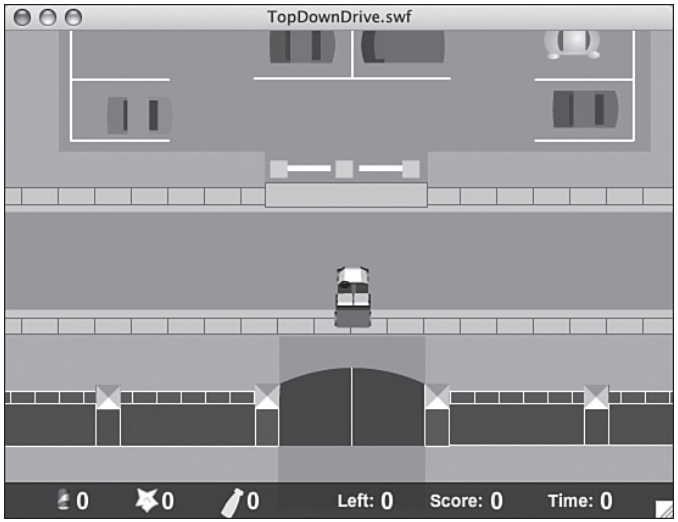


Figure 12.2
Seule une petite zone de 550 × 400 de la carte peut être vue à la fois.

Tous ces éléments graphiques ne sont là que pour la décoration. Ils n'importent pas pour le code du jeu. Voilà une excellente nouvelle pour les graphistes, car cela signifie qu'ils ont toute liberté pour créer l'arrière-plan artistique pour le jeu.

La voiture pourra se déplacer n'importe où dans l'écran avec quelques restrictions simples.

Pour commencer, la voiture sera contrainte de rester dans les limites internes de la clôture. Cette contrainte sera définie par des valeurs `x` et `y` minimales et maximales.

Ensuite, la voiture sera empêchée d'entrer dans la zone de certains autres clips. Nous les appellerons `Blocks`. Si la voiture entre en collision avec l'un de ces `Blocks`, elle sera stoppée à son bord.

Les neuf `Blocks` seront placés sur les neuf blocs de ville dans la carte. La Figure 12.3 signale leurs emplacements avec des bordures épaisses.

Le terme bloc possède trois significations. La plus importante est qu'il bloque la voiture en l'empêchant d'entrer dans la zone. Il représente cependant aussi des blocs d'immeubles dans cette carte. Enfin, il est de forme rectangulaire.

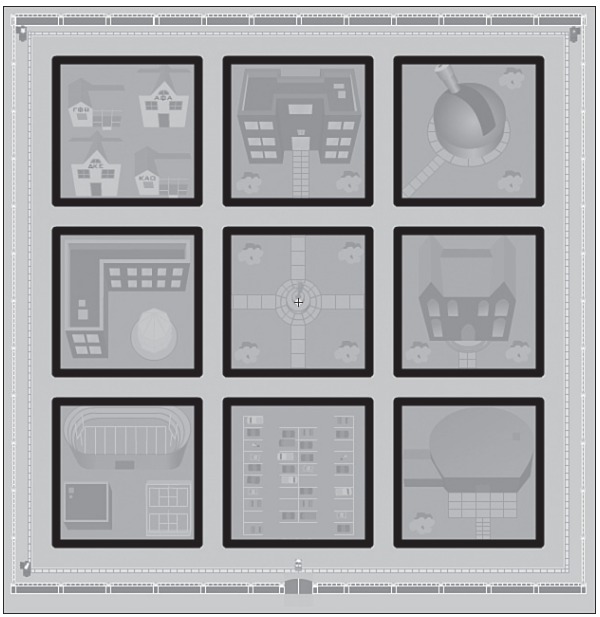


Figure 12.3
Les neuf clips Block sont ici repérés par des bordures épaisses.

Le but du jeu est de récolter les ordures dans le campus et de les déposer dans les bennes de recyclage. Il y aura trois bennes à ordures dans trois des coins du campus.

Il y aura trois types d'ordures, un pour chaque benne : les canettes, les papiers et les bouteilles.

Au lieu de placer manuellement les ordures dans la carte, nous chargerons notre code de le faire. Celui-ci placera cent ordures différentes de manière aléatoire dans le campus. Nous devons nous assurer que ces ordures n'apparaissent pas sur les `Blocks`, sans quoi la voiture ne pourrait les atteindre.

Le défi consiste à ramasser toutes les ordures et à déposer chaque type dans sa propre benne. La voiture ne peut cependant contenir que dix ordures différentes à la fois. Avant de pouvoir en ramasser d'autres, le joueur devra donc commencer par en déposer dans les bennes.

Le jeu se complique ainsi parce que le joueur devra décider le type d'ordure à ramasser en fonction de la benne qu'il souhaite atteindre pour les y déverser.

Conception du jeu

Il vaut la peine d'examiner tous les intrants, les objets et les mécanismes du jeu avant de commencer à programmer notre code. Cela nous permettra de clarifier ce que nous faisons.

Contrôle de la voiture

La voiture sera contrôlée par les touches fléchées. En fait, seules trois des quatre touches fléchées seront requises. La Figure 12.4 présente le clip `car`.

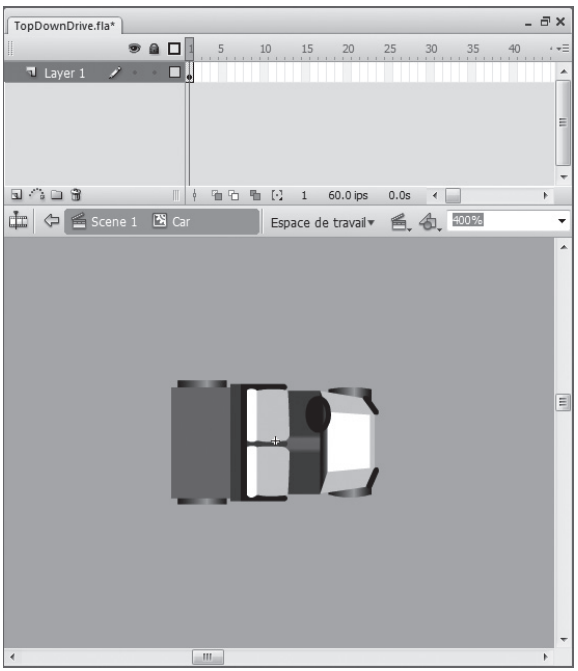


Figure 12.4
Le clip car pointe vers la droite ; valeur rotation 0 correspond donc à la direction que Math.cos et Math.sin représentent.

Nous ne sommes pas en train de créer un jeu de simulation ; nous pouvons donc ignorer l'accélération, le freinage et la marche arrière pour autant que le joueur n'en a pas besoin. Dans le cas présent, il suffit de pouvoir tourner à gauche et à droite et d'avancer pour s'orienter.

Nous utiliserons les touches fléchées de gauche et de droite pour modifier directement la propriété `rotation` de la voiture. Ensuite, nous utiliserons les valeurs `Math.cos` et `Math.sin` de la `rotation` pour définir l'orientation du mouvement en avant. Cette approche s'apparente à celle que nous avons adoptée avec les touches fléchées et la trigonométrie dans le jeu de météorites du Chapitre 7.

Limites de la voiture

La voiture est limitée à rouler sur les rues. Plus exactement, la voiture ne peut pas quitter la carte et ne peut pas rouler dans la zone de l'un des clips `Block`. Le clip `Block` est présenté à la Figure 12.5.

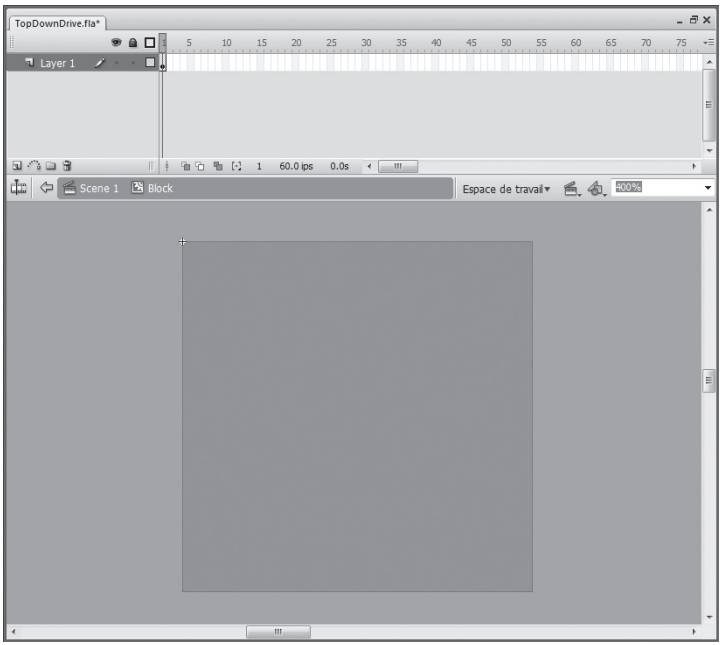


Figure 12.5
Le Block ne se voit jamais dans le jeu ; seuls nous le voyons pendant la conception du niveau. Une fine bordure rouge et un remplissage semi-transparent nous aident à le positionner.

Pour cela, nous comparerons le rectangle de la voiture aux rectangles des objets `Block`. Nous obtiendrons une liste de ces objets lorsque le jeu démarre. Si le rectangle de la voiture et l'un de ceux des `Blocks` se recoupent, nous repousserons la voiture au point où elle se trouve, juste à l'extérieur du `Block`.

Ce mécanisme est semblable à celui que nous avons utilisé pour le jeu de casse-brique du Chapitre 5. Au lieu de faire rebondir la voiture sur le `Block`, nous la positionnerons parfaitement de manière qu'elle se trouve exactement à l'extérieur du `Block`.

Ordures

Les ordures correspondent en fait à un unique clip `TrashObject` de trois images. Nous les placerons de manière aléatoire dans la carte, en nous assurant qu'aucun ne se trouve placé sur les `Blocks`.

Lorsqu'un élément est placé, il est aléatoirement conduit à l'image 1, 2 ou 3, qui représentent chacune un des trois types d'ordures : canettes, papiers ou bouteilles. La Figure 12.6 présente le clip `TrashObject`.

À mesure que la voiture se déplace, nous vérifions si la distance entre chaque `TrashObject` et la voiture est assez réduite pour que la voiture le ramasse.

Nous supprimerons ces objets de l'écran et tiendrons le registre du nombre d'ordures de chaque type ramassé par le joueur. Nous limiterons ce nombre à dix objets à la fois et indiquerons au joueur quand la voiture est pleine.

Lorsque le joueur se rapprochera ensuite suffisamment d'une benne, nous ramènerons à zéro le type d'ordure correspondant dans la collection du joueur. Les joueurs astucieux rempliront leur voiture avec un même type d'ordure et iront déposer les dix éléments dans la benne appropriée.

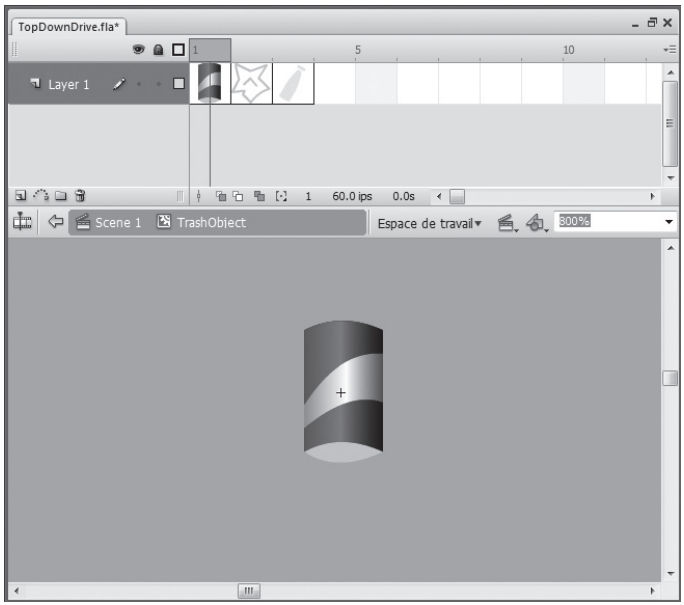


Figure 12.6
Le clip TrashObject contient trois images différentes, chacune avec un type d'ordure différent.

Score et chronomètre du jeu

Les indicateurs de score (voir en bas de la Figure 12.7) sont plus importants dans ce jeu que dans ceux que nous avons créés jusque-là. Le joueur doit y prêter attention.

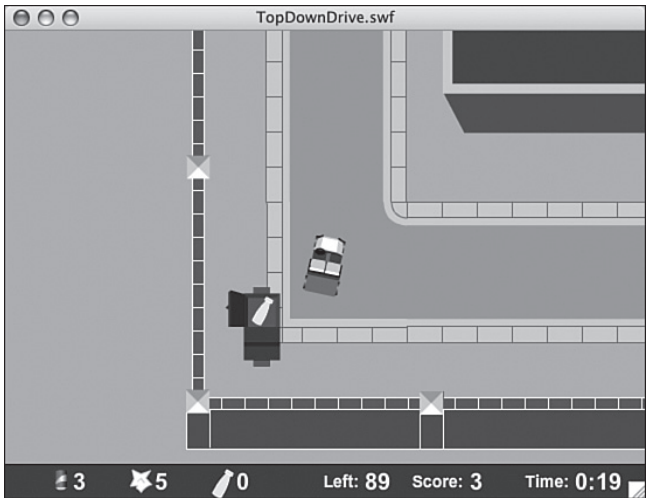


Figure 12.7
Les indicateurs de score se trouvent en bas de l'écran, avec un cadre semi-transparent en dessous.

Les trois premiers indicateurs indiquent le nombre d'ordures que possède le joueur. Comme les joueurs ne peuvent avoir plus de dix ordures avant de se rendre à une benne, ils chercheront de préférence à remplir la voiture avec un type d'ordure particulier. Ils souhaiteront en outre surveiller le moment où leur voiture est presque pleine.

Nous passerons les trois chiffres en rouge lorsque la voiture sera pleine d'ordures. En outre, nous utiliserons un son pour l'indiquer. Un son de collecte s'entendra lorsque le joueur s'approchera suffisamment d'une ordure. Si la voiture est pleine, le joueur entend un autre son et l'ordre reste sur le sol.

Les deux indicateurs suivants présentent le nombre d'ordures qui restent à trouver, le nombre d'ordures ramassées et le temps écoulé. La valeur clé ici est le temps. Les joueurs trouveront tous les cent ordures à moins qu'ils ne se découragent trop tôt. C'est donc le temps qui définira véritablement leur score. Le but du jeu sera de finir le plus vite possible.

La définition de classe

Le code de ce jeu est plutôt simple considérant tout ce que le jeu permet de faire. Le jeu commence en examinant le monde créé dans l'animation Flash et en vérifiant à chaque image les changements et les mouvements du joueur.

Le paquetage commence par importer un grand nombre de bibliothèques de classe. Nous aurons besoin de nos classes habituelles, ainsi que de `flash.geom.*` pour l'utilisation des objets `Point` et `Rectangle` et de `flash.media.Sound` et `flash.media.SoundChannel` pour les effets sonores :

```
package {
    import flash.display.*;
    import flash.events.*;
```

```
import flash.text.*;
import flash.geom.*;
import flash.utils.getTimer;
import flash.media.Sound;
import flash.media.SoundChannel;
```

Le jeu requiert un certain nombre de constantes. `speed` et `turnSpeed` contrôlent la réaction de la voiture aux touches fléchées. `carSize` détermine le rectangle de contour de la voiture par rapport à son point central :

```
public class TopDownDrive extends MovieClip {

    // Constantes
    static const speed:Number = .3;
    static const turnSpeed:Number = .2;
    static const carSize:Number = 50;
```

La constante `mapRect` définit les limites de la carte. Il s'agit approximativement de l'emplacement de la clôture qui entoure le campus :

```
static const mapRect:Rectangle = new Rectangle(-1150,-1150,2300,2300);
```

La constante `numTrashObjects` désigne le nombre d'ordures créées au début du jeu. `maxCarry` définit également le nombre d'ordures que le joueur peut avoir dans la voiture avant de devoir décharger sa cargaison dans une benne :

```
static const numTrashObjects:uint = 100;
static const maxCarry:uint = 10;
```

Les deux constantes suivantes définissent la distance pour les collisions avec les ordures et les bennes. Il se peut que vous deviez ajuster ces valeurs si vous déplacez les bennes un peu à l'écart des routes ou si vous modifiez la constante `carSize` :

```
static const pickupDistance:Number = 30;
static const dropDistance:Number = 40;
```

Il faut éviter de choisir une valeur trop grande pour `pickUpDistance` car il est important que le joueur puisse faire glisser la voiture près des ordures sans les récolter s'il s'efforce de ne ramasser que les ordures d'un certain type.

Les variables peuvent être réparties en trois groupes. Le premier est une série de tableaux qui consignent les objets du jeu.

Le tableau `blocks` contiendra tous les objets `Block` qui empêchent la voiture de quitter la route. `trashObjects` est une liste de toutes les ordures réparties aléatoirement dans la carte. Le tableau `trashcans` contient les trois bennes qui sont les points de dépose pour les ordures :

```
// Objets du jeu
private var blocks:Array;
private var trashObjects:Array;
private var trashcans:Array;
```

L'ensemble de variables suivant concerne l'état du jeu. Nous commencerons par l'habituelle série de variables booléennes des touches fléchées :

```
// Variables de jeu
private var arrowLeft, arrowRight, arrowUp, arrowDown:Boolean;
```

Viennent ensuite deux valeurs de temps. La première, `lastTime`, sera utilisée pour déterminer la durée écoulée depuis la dernière étape d'animation. La seconde, `gameStartTime`, sera utilisée pour déterminer depuis combien de temps la partie a commencé :

```
private var lastTime:int;
private var gameStartTime:int;
```

Le tableau `onboard` est une liste avec un élément pour chaque benne, soit trois éléments. Ils commenceront à 0 et contiendront le nombre de chaque type d'ordure que le joueur transporte dans la voiture :

```
private var onboard:Array;
```

La variable `totalTrashObjects` contiendra la somme des trois nombres dans `onboard`. Nous l'utiliserons en référence rapide lorsqu'il s'agira de décider s'il y a assez de place pour d'autres ordures dans la voiture :

```
private var totalTrashObjects:int;
```

`score` correspond simplement au nombre d'ordures collecté et déposé dans les bennes :

```
private var score:int;
```

La variable `lastObject` est utilisée pour déterminer le moment où le son de charge pleine de la voiture doit être émis. Lorsque le joueur a déjà collecté dix ordures dans sa voiture, nous émettons un son d'alerte, à l'inverse du son positif que le joueur obtient lorsqu'il reste de la place pour les ordures.

Comme l'ordure n'est pas supprimée de la carte, il est fort probable que le joueur entre immédiatement en collision avec elle une nouvelle fois et continue de le faire jusqu'à ce que la voiture s'écarte suffisamment.

Nous enregistrerons donc une référence à l'objet `Trash` dans `lastObject` et la sauvegarderons pour nous y référer par la suite. Nous saurons ainsi que le son négatif a déjà été émis pour cet objet et qu'il n'est pas nécessaire de le produire de nouveau tant que la voiture reste à proximité :

```
private var lastObject:Object;
```

Les dernières variables sont des références aux quatre sons stockés dans la bibliothèque de l'animation. Tous ces sons ont été définis avec des propriétés de liaison qui les mettent à disposition de notre code ActionScript sous forme de classes :

```
// Sons
var theHornSound:HornSound = new HornSound();
var theGotOneSound:GotOneSound = new GotOneSound();
var theFullSound:FullSound = new FullSound();
var theDumpSound:DumpSou nd = new DumpSound();
```

La fonction constructeur

Lorsque l'animation atteint l'image 2, elle appelle `startTopDownDrive` pour commencer le jeu.

Cette fonction appelle immédiatement `findBlocks` et `placeTrash` pour configurer la carte. Nous examinerons ces fonctions sous peu.

```
public function startTopDownDrive() {

    // Récupérer les blocs
    findBlocks();

    // Placer les ordures
    placeTrash();
```

Comme il n'y a que trois bennes et qu'elles ont été spécifiquement nommées dans le `gamesprite`, nous les placerons dans le tableau `trashcans` à l'aide d'une simple ligne de code.

Le `gamesprite` est l'instance sur la scène de l'élément de bibliothèque `GameMap`. Dans la bibliothèque, il s'agit en fait d'un `MovieClip`. Comme il ne fait qu'une image, nous l'appellerons cependant `gamesprite`.

```
// Configuration des bennes
trashcans = new Array(gamesprite.Trashcan1,
    gamesprite.Trashcan2, gamesprite.Trashcan3);
```

Comme les objets `Trash` sont créés par notre code et que la voiture se trouve dans le `gamesprite` avant que notre code s'exécute, les ordures seront au-dessus de la voiture. Cela sera apparent une fois la voiture pleine et que le joueur passe sur d'autres ordures. Si nous n'y faisions rien, vous verriez les ordures flotter au-dessus de la voiture. En appelant `setchildIndex` avec `gamesprite.numChildren-1`, nous replaçons en fait la voiture au-dessus de tous les autres éléments du jeu :

```
// S'assurer que la voiture est au-dessus
gamesprite.setChildIndex(gamesprite.car,gamesprite.numChildren-1);
```

Nous aurions pu aussi créer un clip vide dans le clip `GameMap` afin de contenir toutes les ordures. Nous aurions pu alors le placer dans un calque de scénario juste sous la voiture, au-dessus de la route. Ce serait important si nous souhaitions que des éléments, comme un pont, restent affichés aussi bien au-dessus de la voiture que des ordures.

Il nous faut trois écouteurs, un pour l'événement `ENTER_FRAME`, qui guidera le jeu entier, et deux autres pour les appuis sur les touches :

```
// Ajouter les écouteurs
this.addEventListener(Event.ENTER_FRAME,gameLoop);
stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

Nous configurons ensuite l'état du jeu. `gameStartTime` prend la valeur de temps actuelle. Le tableau `onboard` est rempli avec des zéros, tout comme `totalTrashObjects` et `score` :

```
// Configuration des variables du jeu
gameStartTime = getTimer();
onboard = new Array(0,0,0);
totalTrashObjects = 0;
score = 0;
```

Pour faire avancer le jeu, nous allons immédiatement appeler deux fonctions utilitaires. La fonction `centerMap` permet de positionner le `gamesprite` de manière que la voiture se trouve au centre de l'écran. Si nous ne l'appelions pas maintenant, nous verrions l'espace d'un instant le `gamesprite` tel qu'il apparaît dans le scénario brut avant le premier événement `ENTER_FRAME`.

Un problème du même genre conduit à appeler ici `showScore`, afin que tous les indicateurs de score possèdent leurs valeurs d'origine avant que le joueur ne puisse les voir :

```
centerMap();
showScore();
```

Pour finir, nous émettons un son avec la fonction utilitaire `playSound`. J'ai inclus un son de klaxon simple pour signaler au joueur que le jeu a commencé :

```
        playSound(theHornSound);
    }
```

Trouver les blocs

Pour trouver tous les objets `Block` dans le `gamesprite`, nous devons parcourir en boucle tous les enfants de `gamesprite` et voir lesquels sont de type `Block` à l'aide de l'opérateur `is`. Nous ajouterons ceux qui le sont au tableau `blocks`. Nous positionnerons également la propriété `visible` de chacun des `Blocks` à `false` afin qu'ils ne s'affichent pas pour le joueur. Nous pourrons ainsi les voir

clairement lors du développement de l'animation mais n'aurons pas à nous rappeler de les masquer ou de les rendre transparents avant de finir le jeu :

```
// Trouver tous les objets Block
public function findBlocks() {
    blocks = new Array();
    for(var i=0;i<gamesprite.numChildren;i++) {
        var mc = gamesprite.getChildAt(i);
        if (mc is Block) {
            // Ajouter au tableau et rendre invisible
            blocks.push(mc);
            mc.visible = false;
        }
    }
}
```

Placer les ordures

Pour placer aléatoirement cent ordures, nous devons boucler cent fois en plaçant un objet d'ordure à chaque fois :

```
// Créer des objets Trash aléatoires
public function placeTrash() {
    trashObjects = new Array();
    for(var i:int=0;i<numTrashObjects;i++) {
```

Pour chaque placement, nous commençons une seconde boucle. Ensuite, nous testons différentes valeurs pour les positions `x` et `y` de l'ordure :

```
        // Boucle infinie
        while (true) {

            // Emplacement aléatoire
            var x:Number = Math.floor(Math.random()*mapRect.width)+mapRect.x;
            var y:Number = Math.floor(Math.random()*mapRect.height)+mapRect.y;
```

Une fois que nous avons un emplacement, nous le vérifions par rapport aux `Blocks`. Si l'emplacement se trouve sur un `Block`, nous le notons en donnant à la variable locale `isOnBlock` la valeur `true` :

```
            // Vérifier tous les blocs pour voir si chevauchement
            var isOnBlock:Boolean = false;
            for(var j:int=0;j<blocks.length;j++) {
                if (blocks[j].hitTestPoint(x+gamesprite.x,y+gamesprite.y)) {
                    isOnBlock = true;
                    break;
                }
            }
```

Si l'emplacement n'entre en intersection avec aucun `Block`, nous poursuivons et créons le nouvel objet `TrashObject`. Ensuite, nous définissons son emplacement. Nous avons également besoin de choisir un type aléatoire pour cet élément, en conduisant le clip à l'image 1, 2 ou 3. La Figure 12.8 présente le début d'un jeu où trois clips `TrashObject` ont été placés à proximité du point de départ de la voiture.

Le clip `TrashObject` contient trois images avec un graphisme différent dans chacune. Ces graphismes sont en fait eux-mêmes des clips. Il n'est pas nécessaire qu'il s'agisse de clips séparés pour les utiliser dans `TrashObject`, mais nous souhaitons pouvoir utiliser les mêmes graphismes pour les bennes afin d'indiquer à quel type d'ordure chacune correspond. En procédant ainsi, nous n'avons qu'une version de chaque image dans la bibliothèque.

Nous ajoutons cet objet d'ordure à `trashObjects` et quittons la boucle.

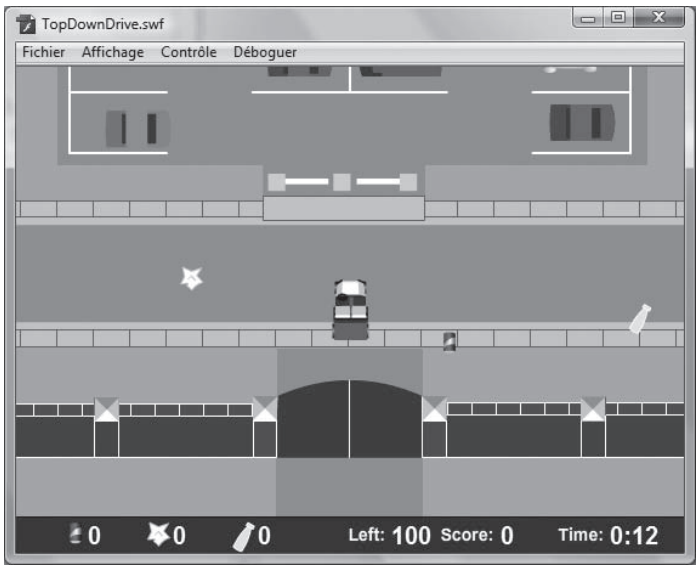


Figure 12.8
Trois clips TrashObject ont été placés aléatoirement près de la voiture au début du jeu.

Cette instruction `break` finale nous fait sortir de la boucle `while` et passer au placement de l'ordure suivante. Si `isOnBlock` vaut `true`, nous poursuivons cependant avec la boucle `while` en choisissant un autre emplacement à tester :

```
// Pas d'intersection, utiliser cet emplacement
if (!isOnBlock) {
    var newObject:TrashObject = new TrashObject();
    newObject.x = x;
    newObject.y = y;
```

```
                newObject.gotoAndStop(Math.floor(Math.random()*3)+1);
                gamesprite.addChild(newObject);
                trashObjects.push(newObject);
                break;
            }
        }
    }
}
```

Lorsque vous testez une fonction de placement comme `placeTrash`, il est utile de la tester avec un nombre d'objets très élevé. Par exemple, j'ai testé `placeTrash` en ayant attribué la valeur 10 000 à `numTrashObjects`. Des tonnes d'ordures se trouvaient ainsi déversées sur la route, mais j'ai pu clairement voir qu'elles ne l'étaient que sur la route et non dans les emplacements où je ne les voulais pas.

Entrée clavier

Le jeu inclut un ensemble de fonctions d'entrée clavier analogues à celles que nous avons utilisées dans plusieurs jeux précédents. Quatre valeurs booléennes sont définies en fonction de l'appui sur les quatre touches fléchées du clavier.

Les fonctions reconnaissent même la touche fléchée du bas bien que cette version du jeu ne l'utilise pas :

```
// Consigner les appuis sur les touches, définir les propriétés
public function keyDownFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        arrowLeft = true;
    } else if (event.keyCode == 39) {
        arrowRight = true;
    } else if (event.keyCode == 38) {
        arrowUp = true;
    } else if (event.keyCode == 40) {
        arrowDown = true;
    }
}
public function keyUpFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        arrowLeft = false;
    } else if (event.keyCode == 39) {
```

```
            arrowRight = false;
        } else if (event.keyCode == 38) {
            arrowUp = false;
        } else if (event.keyCode == 40) {
            arrowDown = false;
        }
    }
```

La boucle du jeu

La fonction `gameLoop` gérera le mouvement de la voiture. Il n'y a en fait pas d'autre objet qui se déplace dans le jeu. Le joueur déplace la voiture et tout le reste est statique dans le `gamesprite`.

Il s'agit d'une animation temporelle ; nous allons donc calculer le temps écoulé depuis la dernière image et déplacer les éléments en fonction de cette valeur :

```
    public function gameLoop(event:Event) {

        // Calculer le temps écoulé
        if (lastTime == 0) lastTime = getTimer();
        var timeDiff:int = getTimer()-lastTime;
        lastTime += timeDiff;
```

Nous vérifierons les touches fléchées de gauche et de droite et appellerons `rotateCar` pour gérer l'orientation de la voiture. Nous passerons `timeDiff` et le sens du tournant :

```
        // Rotation à gauche ou à droite
        if (arrowLeft) {
            rotateCar(timeDiff,"left");
        }
        if (arrowRight) {
            rotateCar(timeDiff,"right");
        }
```

Si la touche fléchée du haut est enfoncée, nous appelons `moveCar` en passant `timeDiff`. Ensuite, nous appelons `centerMap` pour nous assurer que le `gamesprite` est correctement positionné d'après le nouvel emplacement de la voiture.

La fonction `checkCollisions` vérifie si le joueur a collecté des ordures ou s'est approché d'une benne :

```
        // Déplacer la voiture
        if (arrowUp) {
            moveCar(timeDiff);
            centerMap();
            checkCollisions();
        }
```

Rappelez-vous que le temps est en fait le véritable score de ce jeu. Le joueur joue contre la montre. Nous devons donc mettre à jour le temps écoulé afin que le joueur puisse évaluer sa réussite :

```
        // Mettre à jour le temps et vérifier si le jeu est terminé
        showTime();
    }
```

Examinons tout de suite la fonction `centerMap`, parce qu'elle est particulièrement simple. Elle doit juste définir l'emplacement du `gamesprite` en lui attribuant les versions négatives de l'emplacement de la voiture à l'intérieur du `gamesprite`. Par exemple, si la voiture se trouve à l'emplacement `1000,600` dans `gamesprite`, le fait de positionner le `gamesprite` à `-1000,-600` implique que la voiture se trouvera à l'emplacement `0,0` dans la scène.

Nous ne souhaitons pas que la voiture soit précisément à `0,0`, car cela correspond au coin supérieur gauche de la scène. Nous souhaitons la centrer dans la scène. Pour cela, nous ajoutons donc `275,200`.

Si vous souhaitez modifier la taille de la zone visible de la scène, par exemple pour passer à 640 × 480, il est nécessaire de changer également les valeurs à cet endroit pour les faire correspondre au milieu de la scène. Une scène de 640 × 480 nécessiterait les valeurs `320` et `240` pour les ajustements `x` et `y` afin de positionner la voiture au milieu de l'écran.

```
public function centerMap() {
    gamesprite.x = -gamesprite.car.x + 275;
    gamesprite.y = -gamesprite.car.y + 200;
}
```

Mouvement de la voiture

Le guidage de la voiture est assez peu réaliste dans ce jeu : la voiture pivote autour de son centre de quelques degrés à chaque image. En fait, la voiture peut même tourner sans avancer. Essayez donc avec votre propre voiture…

Si vous jouez à ce jeu, vous remarquerez cependant à peine cette incongruité. La rotation est temporelle ; elle est donc le produit de `timeDiff` et de la constante `turnSpeed`. La voiture doit tourner à la même cadence quelle que soit la cadence d'images de l'animation :

```
public function rotateCar(timeDiff:Number, direction:String) {
    if (direction == "left") {
        gamesprite.car.rotation -= turnSpeed*timeDiff;
    } else if (direction == "right") {
        gamesprite.car.rotation += turnSpeed*timeDiff;
    }
}
```

Il est aussi assez simple de faire avancer la voiture ou, du moins, cela le serait s'il ne fallait détecter et gérer les collisions avec les `Blocks` et les limites de la carte. Nous simplifierons la détection des collisions en utilisant des objets `Rectangle` simples et la fonction `intersects`. La première chose dont nous ayons besoin est donc le `Rectangle` de la voiture.

La voiture est déjà de forme rectangulaire mais, comme elle pivote, il est problématique d'utiliser le `Rectangle` exact du clip. Au lieu de cela, nous allons utiliser un `Rectangle` recréé qui utilise le centre de la voiture et `carSize`. Cette zone carrée offrira une approximation suffisante de la zone de la voiture pour que le joueur ne s'en rende pas compte.

Pour préserver l'illusion que les collisions sont précises, il est important de veiller à ce que le graphisme de la voiture soit relativement carré, autrement dit aussi long que large. Une voiture bien plus longue que large nécessiterait que l'on fasse dépendre la distance de collision de la rotation de la voiture par rapport aux bords avec lesquels elle pourrait entrer en collision. Ce cas de figure serait déjà bien plus complexe.

```
// Faire avancer la voiture
public function moveCar(timeDiff:Number) {
    // Calculer la zone actuelle de la voiture
    var carRect = new Rectangle(gamesprite.car.x-carSize/2,
        gamesprite.car.y-carSize/2, carSize, carSize);
```

Nous avons maintenant l'emplacement actuel de la voiture dans `carRect`. Pour calculer le nouvel emplacement de la voiture, nous convertissons la rotation de la voiture en radians, fournissons ces nombres à `Math.cos` et `Math.sin`, puis multiplions ces valeurs par la vitesse et `timeDiff`. Nous obtenons ainsi un mouvement temporel en utilisant la constante `speed`. `newCarRect` contient ensuite le nouvel emplacement de la voiture :

```
    // Calculer la nouvelle zone de la voiture
    var newCarRect = carRect.clone();
    var carAngle:Number = (gamesprite.car.rotation/360)*(2.0*Math.PI);
    var dx:Number = Math.cos(carAngle);
    var dy:Number = Math.sin(carAngle);
    newCarRect.x += dx*speed*timeDiff;
    newCarRect.y += dy*speed*timeDiff;
```

Nous avons également besoin des emplacements `x` et `y` correspondant au nouveau `Rectangle`. Nous ajouterons les mêmes valeurs à `x` et à `y` pour obtenir ce nouvel emplacement :

```
    // Calculer le nouvel emplacement
    var newX:Number = gamesprite.car.x + dx*speed*timeDiff;
    var newY:Number = gamesprite.car.y + dy*speed*timeDiff;
```

Il est maintenant temps de parcourir en boucle les blocs et de voir si le nouvel emplacement entre en intersection avec l'un d'entre eux :

```
// Parcourir en boucle les blocs et vérifier les collisions
for(var i:int=0;i<blocks.length;i++) {

    // Récupérer le rectangle du bloc, voir s'il y a collision
    var blockRect:Rectangle = blocks[i].getRect(gamesprite);
    if (blockRect.intersects(newCarRect)) {
```

En cas de collision, nous examinons séparément les données horizontales et verticales de la collision.

Si la voiture a passé le côté gauche d'un `Block`, nous la ramenons au bord de ce `Block`. Le même principe est utilisé pour le côté droit du `Block`. Nous n'avons pas à nous soucier d'ajuster le `Rectangle`, mais uniquement les valeurs de position `newX` et `newY`. Ce sont elles qui seront utilisées pour définir le nouvel emplacement de la voiture :

```
        // Repousser horizontalement la voiture
        if (carRect.right <= blockRect.left) {
            newX += blockRect.left - newCarRect.right;
        } else if (carRect.left >= blockRect.right) {
            newX += blockRect.right - newCarRect.left;
        }
```

Voici maintenant le code qui gère les côtés supérieur et inférieur du `Block` heurté :

```
        // Repousser verticalement la voiture
        if (carRect.top >= blockRect.bottom) {
            newY += blockRect.bottom-newCarRect.top;
        } else if (carRect.bottom <= blockRect.top) {
            newY += blockRect.top - newCarRect.bottom;
        }
    }

}
```

Une fois que tous les blocs ont été examinés afin de déterminer les éventuelles collisions, nous devons examiner les limites de la carte. C'est l'opposé des blocs, car nous souhaitons conserver la voiture à l'intérieur du `Rectangle` des limites et non en dehors.

Nous examinerons donc chacun des quatre côtés et repousserons les valeurs `newX` ou `newY` afin d'empêcher la voiture de quitter les limites de la carte :

```
// Vérifier les collisions avec les bords
if ((newCarRect.right > mapRect.right) && (carRect.right <= mapRect.right)) {
    newX += mapRect.right - newCarRect.right;
```

```
        }
        if ((newCarRect.left < mapRect.left) && (carRect.left >= mapRect.left)) {
            newX += mapRect.left - newCarRect.left;
        }

        if ((newCarRect.top < mapRect.top) && (carRect.top >= mapRect.top)) {
            newY += mapRect.top-newCarRect.top;
        }
        if ((newCarRect.bottom > mapRect.bottom) && (carRect.bottom <= mapRect.bottom)) {
            newY += mapRect.bottom - newCarRect.bottom;
        }
```

Maintenant que la voiture est sécurisée dans les limites de la carte et rejetée hors de tout `Block`, nous pouvons définir le nouvel emplacement de la voiture :

```
        // Définir le nouvel emplacement de la voiture
        gamesprite.car.x = newX;
        gamesprite.car.y = newY;
    }
```

Vérifier les collisions avec les ordures et les bennes

La fonction `checkCollisions` doit rechercher deux types de collisions différents. Elle commence par examiner tous les `trashObjects`. Elle utilise la fonction `Point.distance` pour voir si l'emplacement de la voiture et celui du `TrashObject` sont plus rapprochés que la constante `pickupDistance` :

```
    public function checkCollisions() {

        // Parcourir les bennes en boucle
        for(var i:int=trashObjects.length-1;i>=0;i--) {

            // Voir si proximité suffisante pour récupérer les ordures
            if (Point.distance(new Point(gamesprite.car.x,gamesprite.car.y),
                new Point(trashObjects[i].x, trashObjects[i].y)) < pickupDistance) {
```

Si un élément est suffisamment proche, nous comparons `totalTrashObjects` à la constante `maxCarry`. S'il reste de la place, l'ordure est ramassée en positionnant l'emplacement approprié dans `onboard` d'après `currentFrame-1` du clip `TrashObject`. Ensuite, l'ordure est supprimée de `gamesprite` et du tableau `trashObjects`. Nous devons mettre à jour le score et lire `GotOneSound` :

```
                // Voir s'il y a de la place
                if (totalTrashObjects < maxCarry) {
                    // Récupérer l'ordure
```

```
                onboard[trashObjects[i].currentFrame-1]++;
                gamesprite.removeChild(trashObjects[i]);
                trashObjects.splice(i,1);
                showScore();
                playSound(theGotOneSound);
```

L'un des aspects de notre code susceptible de porter à confusion tient à la manière dont les types d'ordures sont référencés. Comme images dans le clip `TrashObject`*, ils correspondent aux images 1, 2 et 3. En revanche, les tableaux sont indicés à 0, aussi, dans le tableau* `onboard`*, nous stockons les types d'ordures 1, 2 et 3 dans les emplacements de Tableau 0, 1 et 2. Les bennes seront nommées* `Trashcan1`*,* `Trashcan2` *et* `Trashcan3` *et correspondront aux numéros des images, mais pas aux emplacements de tableau. Pour autant que vous gardiez ce point à l'esprit, vous ne devriez pas avoir de problème pour modifier le code. Le fait que les tableaux soient indicés à 0 et que les images commencent à 1 soulève des problèmes constants pour les développeurs ActionScript.*

À l'inverse, si le joueur entre en collision avec une ordure mais qu'il n'y ait plus de place, nous émettons un autre son. Nous ne le faisons jouer que si l'élément ne correspond pas à `lastObject` (le dernier objet). Cette vérification évite que le son ne se joue de manière répétitive le temps que le joueur s'écarte de l'objet. Le son n'est ainsi joué qu'une seule fois par objet :

```
            } else if (trashObjects[i] != lastObject) {
                playSound(theFullSound);
                lastObject = trashObjects[i];
            }
        }
    }
```

L'ensemble suivant de collisions examine les trois bennes. Nous utiliserons ici aussi `Point.distance`. Après qu'une collision aura été détectée, nous supprimerons toutes les ordures de ce type du tableau `onboard`. Nous mettrons à jour le score et émettrons un son signalant que les ordures sont déposées :

```
// Déposer les ordures si proches d'une benne
for(i=0;i<trashcans.length;i++) {

    // Voir si la voiture est suffisamment proche
    if (Point.distance(new Point(gamesprite.car.x,gamesprite.car.y),
        new Point(trashcans[i].x, trashcans[i].y)) < dropDistance) {

        // Voir si le joueur a des ordures de ce type
        if (onboard[i] > 0) {
```

```
                    // Déposer
                    score += onboard[i];
                    onboard[i] = 0;
                    showScore();
                    playSound(theDumpSound);
```

Si le score s'est élevé au point d'atteindre la valeur de la constante `numTrashObjects`, nous en concluons que la dernière ordure a été déposée et la partie est terminée :

```
                    // Voir si toutes les ordures ont été déposées
                    if (score >= numTrashObjects) {
                        endGame();
                        break;
                    }
                }
            }
        }
    }
```

Le chronomètre

La mise à jour du chronomètre est assez simple et analogue à ce que nous avons fait dans le jeu de Memory du Chapitre 3. Nous soustrayons le temps courant du temps de départ pour obtenir le nombre de millisecondes écoulées depuis le début du jeu. Ensuite, nous utilisons la fonction utilitaire `clockTime` pour convertir cette valeur en un format d'horloge :

```
// Mettre à jour le temps affiché
public function showTime() {
    var gameTime:int = getTimer()-gameStartTime;
    timeDisplay.text = clockTime(gameTime);
}
```

La fonction `clockTime` calcule le nombre de secondes et de minutes, puis formate ces valeurs avec des zéros d'en-tête si besoin :

```
// Conversion au format d'horloge
public function clockTime(ms:int):String {
    var seconds:int = Math.floor(ms/1000);
    var minutes:int = Math.floor(seconds/60);
    seconds -= minutes*60;
    var timeString:String = minutes+":"+String(seconds+100).substr(1,2);
    return timeString;
}
```

Les indicateurs de score

Dans ce jeu, l'affichage du score est complexe et ne se limite pas à afficher un simple nombre. Nous allons afficher les trois nombres stockés dans `onboard`. Dans le même temps, nous additionnerons ces nombres pour obtenir la valeur `totalTrashObjects`, qui sera utilisée autre part dans le jeu afin de déterminer s'il reste de la place dans la voiture :

```
// Mettre à jour les éléments texte du score
public function showScore() {

    // Définir chaque nombre d'ordure, additionner le tout
    totalTrashObjects = 0;
    for(var i:int=0;i<3;i++) {
        this["onboard"+(i+1)].text = String(onboard[i]);
        totalTrashObjects += onboard[i];
    }
```

Nous utiliserons également `totalTrashObjects` dès maintenant pour colorer les trois nombres en rouge ou en blanc selon que la voiture est pleine ou non. Ce formatage offrira un indicateur naturel au joueur afin de lui indiquer qu'il a atteint la capacité maximale de la voiture et doit se rendre à une benne :

```
    // Définir la couleur des trois en fonction de la limite atteinte
    for(i=0;i<3;i++) {
        if (totalTrashObjects >= 10) {
            this["onboard"+(i+1)].textColor = 0xFF0000;
        } else {
            this["onboard"+(i+1)].textColor = 0xFFFFFF;
        }
    }
```

Ensuite, nous affichons à la fois le score et le nombre d'ordures restant à ramasser :

```
    // Définir le nombre restant et le score
    numLeft.text = String(trashObjects.length);
    scoreDisplay.text = String(score);
}
```

Fin du jeu

Lorsque la partie est terminée, nous supprimons les écouteurs mais pas le `gamesprite`, car nous ne l'avons pas créé. Celui-ci disparaît simplement lorsque nous utilisons `gotoAndStop` pour nous rendre à l'image suivante.

Comme le gamesprite ne se trouve que dans l'image play, il n'est pas affiché dans l'image gameover :

```
// Fin de la partie, supprimer les écouteurs
public function endGame() {
    blocks = null;
    trashObjects = null;
    trashcans = null;
    this.removeEventListener(Event.ENTER_FRAME,gameLoop);
    stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
    stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
    gotoAndStop("gameover");
}
```

Lorsque l'image gameover a été atteinte, nous appelons showFinalMessage. Nous ne pouvons l'appeler avant car le champ texte finalMessage ne se trouve que dans l'image gameover et n'est accessible qu'une fois que cette image est visible.

Nous placerons le temps final dans ce champ texte :

```
// Afficher le temps dans l'écran final
public function showFinalMessage() {
    showTime();
    var finalDisplay:String = "";
    finalDisplay += "Time: "+timeDisplay.text+"\n";
    finalMessage.text = finalDisplay;
}
```

Il nous manque une dernière fonction : la fonction utilitaire playSound. Celle-ci sert simplement d'emplacement central pour tous les effets sons à déclencher :

```
public function playSound(soundObject:Object) {
    var channel:SoundChannel = soundObject.play();
}
```

L'un des avantages liés au fait d'utiliser une même fonction partout où les effets son sont initiés tient à ce que vous pouvez rapidement et aisément créer des fonctions de sourdine et de volume. Si vous éparpillez votre code de son dans tout le jeu, vous serez contraint de modifier chacun de ces emplacements pour ajouter un système de sourdine ou de réglage du volume.

Modifier le jeu

Ce jeu peut être modifié pour correspondre à presque n'importe quel type de jeu d'exploration ou de collecte d'objets. Vous pouvez modifier les éléments d'arrière-plan sans aucune programmation particulière. Les zones de collision (les `Blocks`) peuvent être modifiés en déplaçant et en ajoutant simplement de nouveaux clips `Block`.

Vous pouvez même faire durer le jeu plus longtemps en faisant apparaître de nouvelles ordures à mesure que le temps passe. Vous pouvez par exemple configurer un `Timer` de façon qu'une nouvelle ordure soit ajoutée toutes les cinq secondes. Le `Timer` pourrait procéder de cette manière pendant quelques minutes avant de s'interrompre finalement.

Vous pourriez également ajouter des obstacles à éviter, comme des tâches d'huile ou des mines. Une version militaire de ce jeu pourrait ainsi mettre en scène un véhicule militaire chargé de récupérer des soldats sur un champ de bataille tout en évitant les mines qui jonchent le sol.

Créer un jeu de course

En jouant à notre jeu en vue aérienne, vous serez peut-être tenté de vous amuser à faire la course de voiture. Vous pourriez ainsi essayer de voir le temps qu'il vous faut pour faire le tour du campus.

Si la précédente version du jeu constitue un bon départ, il convient tout de même d'ajouter quelques éléments supplémentaires pour créer un jeu de course.

Codes sources

http://flashgameu.com

A3GPU12_RacingGame.zip

Éléments du jeu de course

Bien que nous créions un jeu de course "d'arcade" et non un véritable jeu de simulation, il convient de rendre la conduite suffisamment réaliste pour donner l'impression que l'on pilote une vraie voiture. Il ne faut donc pas que la voiture se retrouve lancée à pleine vitesse dès la première seconde où l'on appuie sur la touche ni qu'elle s'arrête dès l'instant où on la relâche.

Nous allons donc ajouter des effets d'accélération et de décélération à ce jeu. La touche fléchée du haut contribuera à accélérer la vitesse de la voiture et la vitesse de la voiture sera utilisée pour déterminer le mouvement à chaque image.

La distinction entre un jeu d'arcade et un jeu de simulation est plus importante ici que dans aucun des jeux que nous avons considérés précédemment. Une véritable simulation doit tenir compte de la réalité physique des éléments, comme la masse de la voiture, le couple moteur du véhicule et la friction entre les pneus et la route, sans mentionner encore les dérapages. Ces points de détail dépassent non seulement le cadre d'un jeu simple créé sous Flash mais sont généralement simplifiés ou ignorés dans bien des jeux de console coûteux. Il est important de ne pas laisser le réalisme empiéter sur le plaisir du jeu ni de venir entraver le bon déroulement du développement du jeu en abandonnant sa réalisation faute de budget et de temps.

De la même manière, si la touche fléchée du bas est enfoncée, une accélération inverse s'opère. En position d'arrêt, la touche fléchée du bas produira donc une valeur de vitesse négative et fera reculer la voiture.

L'un des autres aspects du jeu de course tient à ce que la voiture doit suivre un tracé spécifique. Le joueur ne doit pas pouvoir couper la piste en opérant un raccourci ni faire machine arrière pour retraverser la ligne d'arrivée en quelques secondes.

Pour surveiller le cheminement du joueur, nous utiliserons une technique simple dite des points de parcours. Le joueur doit alors se rapprocher d'une série d'emplacements autour de la piste et récolter les points associés. Seul le joueur qui a atteint tous ces points est autorisé à passer la ligne d'arrivée.

Le plus intéressant concernant les points de parcours tient à ce que le joueur ne sait même pas où ils se trouvent. Nous masquerons ces points et comptabiliserons le score du parcours sans en informer le joueur. Ce dernier ne saura en définitive qu'une chose : qu'il doit courir vite et honnêtement.

Une autre fonctionnalité de ce jeu contribuera à le rendre plus palpitant : le compte à rebours de départ. Au lieu de faire commencer le jeu directement, nous bloquerons le joueur pendant 3 secondes en affichant 3, puis 2 puis 1 avant le top départ.

Créer la piste

La détection de collision dans le jeu en vue aérienne utilisait des blocs rectangulaires. Il est assez facile de détecter des collisions par rapport à des bords horizontaux ou verticaux droits.

Les pistes de courses présentent en revanche des courbes, et la détection des collisions avec des courbes, voire simplement avec de courts segments de murs en diagonale, se révèle bien plus difficile.

Nous éluderons donc ce problème dans ce jeu.

La piste sera constituée de trois zones : la route, les bordures et tout le reste. Si la voiture se trouve sur la route, elle se déplace sans contrainte. Si elle se trouve sur la bordure de la route, elle continue de se déplacer, mais avec une enquiquinante décélération constante qui fera perdre du temps au joueur. Si la voiture se trouve en dehors de la route et de sa bordure, la décélération deviendra drastique et la voiture devra tourner et boitiller jusqu'à la route.

La Figure 12.9 présente ces trois zones. La route se trouve au milieu et apparaît en gris dans Flash. Juste à l'extérieur se trouve sa bordure représentée en marron dans Flash et d'un ton de gris différent dans la figure.

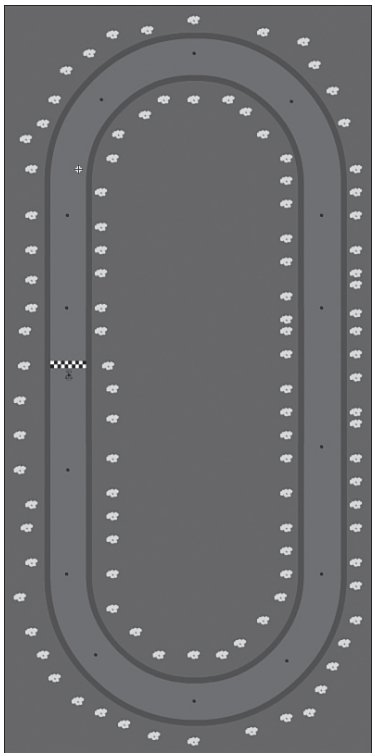


Figure 12.9
La piste est entourée d'une bordure épaisse.

La piste inclut également des éléments inactifs comme les arbres éparpillés autour.

Si les arbres ne sont pas référencés dans notre code et ne sont même pas des clips mais simplement des symboles graphiques, ils jouent néanmoins un rôle important. Sans ces éléments incidents, il serait difficile pour le joueur de remarquer le mouvement de la voiture et d'évaluer sa vitesse.

Le clip `car` est placé sur la piste à la position où la voiture doit démarrer, soit exactement sur la ligne d'arrivée, qui correspond à un clip séparé.

Les points qui apparaissent autour de la piste sont les points de parcours. Vous pouvez n'en placer que quelques-uns autour de la piste, comme nous l'avons fait, ou bien plus si cette piste inclut plus de tournants et de chicanes et que vous deviez éviter que le joueur triche en coupant les virages.

Tous ces éléments se trouvent dans le clip `Track`, qui est le `gamesprite` auquel notre code fait référence.

Effets sonores

Ce jeu utilisera plusieurs effets sonores. Trois différents sons de conduite seront lus en boucle pendant que le joueur fait avancer la voiture. Voici une liste des sons utilisés dans le jeu :

- **`DriveSound`**. Un son en boucle émis pendant que la voiture accélère et se trouve sur la route. Analogue au son d'un moteur de voiture de sport.
- **`SideSound`.** Un son en boucle émis pendant que la voiture accélère et se trouve sur la bordure de la route. Analogue au son de pneus qui patinent dans la boue.
- **`OffroadSound`.** Un son en boucle émis pendant que la voiture accélère et se trouve hors de la route et de sa bordure. Analogue au son d'une voiture qui roule sur des gravillons.
- **`BrakestopSound`.** Un son de freins crissant à utiliser lorsque la voiture passe la ligne d'arrivée.
- **`ReadysetSound`.** Un bip aigu émis durant le compte à rebours au début du jeu.
- **`GoSound`.** Un bip grave émis lorsque le compte à rebours atteint zéro.

Le jeu pourrait facilement inclure d'autres sons, comme un son dormant lorsque la voiture n'accélère pas. `BrakestopSound` pourrait en outre être remplacé par une foule qui acclame le joueur à la fin de la course.

Constantes et variables

Certaines parties du code de ce jeu sont identiques à celles du jeu de conduite en vue aérienne. Nous nous occuperons ici avant tout du code qui change.

Les constantes incluent maintenant des constantes d'accélération et de décélération. Elles correspondent à des nombres particulièrement petits parce qu'elles seront multipliées par les millisecondes qui s'écoulent entre les images :

```
public class Racing extends MovieClip {

    // Constantes
    static const maxSpeed:Number = .3;
    static const accel:Number = .0002;
    static const decel:Number = .0003;
    static const turnSpeed:Number = .18;
```

Parmi les variables du jeu figure `gameMode`, qui indiquera si la course a démarré. Nous aurons également un tableau `waypoints` pour contenir les emplacements `Point` des clips `Waypoint`. La variable `speed` contiendra la cadence actuelle à laquelle la piste se déplace, qui changera à mesure que la voiture accélère et décélère :

```
    // Variables de jeu
    private var arrowLeft, arrowRight, arrowUp, arrowDown:Boolean;
    private var lastTime:int;
```

```
        private var gameStartTime:int;
        private var speed:Number;
        private var gameMode:String;
        private var waypoints:Array;
        private var currentSound:Object;
```

Voici les définitions initiales pour tous les nouveaux sons. Chacun se trouve dans la bibliothèque et a été configuré de manière à être exporté pour ActionScript :

```
        // Sons
        static const theBrakestopSound:BrakestopSound = new BrakestopSound();
        static const theDriveSound:DriveSound = new DriveSound();
        static const theGoSound:GoSound = new GoSound();
        static const theOffroadSound:OffroadSound = new OffroadSound();
        static const theReadysetSound:ReadysetSound = new ReadysetSound();
        static const theSideSound:SideSound = new SideSound();
        private var driveSoundChannel:SoundChannel;
```

Démarrer le jeu

Lorsque ce jeu démarre, il n'a pas besoin de rechercher des `Blocks`. Il doit au lieu de cela trouver les points de parcours (`Waypoints`). La fonction `findWaypoints` s'en charge. Nous y viendrons juste après :

```
    public function startRacing() {

        // Obtenir la liste des points de parcours
        findWaypoints();
```

Les écouteurs requis sont les mêmes que pour le jeu de conduite en vue aérienne, mais parmi les variables qui doivent être définies au début du jeu figurent maintenant `gameMode` et `speed`. Nous définirons également le champ texte `timeDisplay` en lui attribuant une chaîne vide parce qu'il sera vide pendant les 3 premières secondes du jeu, jusqu'à ce que la course démarre :

```
        // Ajouter les écouteurs
        this.addEventListener(Event.ENTER_FRAME,gameLoop);
        stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);

        // Configurer les variables du jeu
        speed = 0;
        gameMode = "wait";
        timeDisplay.text = "";
        gameStartTime = getTimer()+3000;
        centerMap();
    }
```

Vous remarquerez que `gameStartTime` se voit ajouter trois secondes. En effet, le jeu démarre avec un compte à rebours de 3 secondes. La voiture ne sera pas autorisée à se déplacer avant que ces 3 secondes se soient écoulées et que `gameTimer()` ait rattrapé `gameStartTime`.

La fonction `findWaypoints` est très proche de la fonction `findBlocks` du précédent jeu. Cette fois, nous ne souhaitons cependant connaître que l'emplacement du `Point` de chaque point de parcours. Une fois que nous avons enregistré cette information, le clip n'importe plus :

```
// Examiner tous les enfants de gamesprite et mémoriser les points de parcours
public function findWaypoints() {
    waypoints = new Array();
    for(var i=0;i<gamesprite.numChildren;i++) {
        var mc = gamesprite.getChildAt(i);
        if (mc is Waypoint) {
            // Ajouter au tableau et rendre invisible
            waypoints.push(new Point(mc.x, mc.y));
            mc.visible = false;
        }
    }
}
```

La boucle principale du jeu

Nous passerons les fonctions d'écouteur clavier parce qu'elles sont identiques à celles du jeu précédent. La fonction `gameLoop` est en revanche un peu différente. Nous y inclurons directement un grand nombre des mécanismes du jeu au lieu de les déléguer à d'autres fonctions.

Après avoir déterminé le temps qui s'est écoulé depuis la dernière exécution de `gameLoop`, nous examinerons les touches fléchées de gauche et de droite et tournerons la voiture :

```
public function gameLoop(event:Event) {

    // Calculer le temps écoulé
    if (lastTime == 0) lastTime = getTimer();
    var timeDiff:int = getTimer()-lastTime;
    lastTime += timeDiff;

    // Ne déplacer la voiture qu'en mode Course
    if (gameMode == "race") {
        // Pivoter à gauche ou à droite
        if (arrowLeft) {
            gamesprite.car.rotation -= (speed+.1)*turnSpeed*timeDiff;
        }
        if (arrowRight) {
            gamesprite.car.rotation += (speed+.1)*turnSpeed*timeDiff;
        }
```

Trois facteurs affectent la valeur de rotation : la vitesse (`speed`), la constante `turnSpeed` (braquage) et `timeDiff`. `speed` est en outre augmentée de `.1`. Ce réglage permet au joueur de tourner la voiture légèrement lorsqu'il se trouve à l'arrêt et légèrement plus lorsqu'il se déplace lentement. Si cela ne correspond pas précisément à une simulation de conduite, cela rend néanmoins le jeu moins frustrant.

En liant la vitesse au braquage, nous permettons à la voiture de tourner plus vite lorsqu'elle se déplace plus vite. Cela rend la conduite plus réaliste et permet de mieux gérer les courbes du circuit.

En outre, vous noterez que la rotation et le mouvement qui suit ne se produisent que si `gameMode` vaut `"race"`. Cela ne se produit qu'une fois le compte à rebours de 3 secondes écoulé.

Le mouvement de la voiture dépend de sa vitesse. La vitesse dépend de l'accélération, qui se produit lorsque le joueur appuie sur les touches fléchées du haut et du bas. Le code qui suit se charge de ces changements et s'assure que la vitesse ne s'emballe pas en la restreignant à `maxSpeed` :

```
// Faire accélérer la voiture
if (arrowUp) {
    speed += accel*timeDiff;
    if (speed > maxSpeed) speed = maxSpeed;
} else if (arrowDown) {
    speed -= accel*timeDiff;
    if (speed < -maxSpeed) speed = -maxSpeed;
```

Si ni la touche fléchée du haut ni celle du bas ne sont enfoncées, la voiture doit s'arrêter progressivement. Nous utiliserons la constante `decel` pour réduire la vitesse de la voiture :

```
// Aucune touche enfoncée, ralentir
} else if (speed > 0) {
    speed -= decel*timeDiff;
    if (speed < 0) speed = 0;
} else if (speed < 0) {
    speed += decel*timeDiff;
    if (speed > 0) speed = 0;
}
```

Vous pourriez aisément ajouter des freins à la voiture. Pour cela, incluez la barre d'espace en plus des quatre touches fléchées lorsque vous surveillez le clavier. Lors de l'appui sur la barre d'espace, vous pouvez provoquer un ralentissement plus important que la constante `decel`.

Nous n'avons à vérifier le mouvement de la voiture que s'il y a une valeur `speed`. Si la voiture est complètement à l'arrêt, nous pouvons ignorer ce qui suit.

Si la voiture se déplace, nous devons en revanche la repositionner, vérifier si elle se trouve ou non sur la route, recentrer la carte par rapport à la voiture, vérifier si de nouveaux points de parcours ont été rencontrés et vérifier si la voiture a franchi la ligne d'arrivée :

```
            // En cas de déplacement, bouger la voiture et vérifier l'état
            if (speed != 0) {
                moveCar(timeDiff);
                centerMap();
                checkWaypoints();
                checkFinishLine();
            }
        }
```

Que la voiture bouge ou non, le chronomètre doit pour sa part être mis à jour :

```
        // Mettre à jour le temps et vérifier si la course est terminée
        showTime();
    }
```

Mouvement de la voiture

La voiture se déplace en fonction de `rotation`, `speed` et `timeDiff`. La `rotation` est convertie en radians et la valeur, fournie à `Math.cos` et `Math.sin`. La position originale de la voiture est stockée dans `carPos` et le changement de position, dans `dx` et `dy` :

```
    public function moveCar(timeDiff:Number) {

        // Obtenir la position actuelle
        var carPos:Point = new Point(gamesprite.car.x, gamesprite.car.y);

        // Calculer le changement
        var carAngle:Number = gamesprite.car.rotation;
        var carAngleRadians:Number = (carAngle/360)*(2.0*Math.PI);
        var carMove:Number = speed*timeDiff;
        var dx:Number = sMath.cos(carAngleRadians)*carMove;
        var dy:Number = Math.sin(carAngleRadians)*carMove;
```

Pendant que nous déterminons où doit se trouver le nouvel emplacement de la voiture, nous devons aussi déterminer le son qui doit être joué. Si la voiture se déplace et se trouve sur la route, il faut lire le son `theDriveSound`.

Nous supposerons que c'est le cas à ce stade et ajusterons la valeur de `newSound` à mesure que nous examinerons d'autres aspects de l'état du jeu :

```
// Nous supposons que nous allons utiliser le son de conduite
var newSound:Object = theDriveSound;
```

Le premier test à réaliser ici consiste à voir si la voiture se trouve actuellement sur la route. Nous utiliserons `hitTestPoint` pour cela. Le troisième paramètre dans `hitTestPoint` nous permet de tester un point par rapport à la forme spécifique de la route. Nous devons ajouter `gamesprite.x` et `gamesprite.y` à la position de la voiture, car `hitTestPoint` fonctionne au niveau de la scène, avec les positions de la scène et non au niveau du `gamesprite` avec les positions du `gamesprite` :

```
// Voir si la voiture n'est PAS sur la route
if (!gamesprite.road.hitTestPoint(carPos.x+dx+gamesprite.x,
    carPos.y+dy+gamesprite.y, true)) {
```

Notez la présence du point d'exclamation crucial dans la ligne de code précédente. Ce ! signifie "non" et inverse la valeur booléenne qui le suit. Au lieu de vérifier si l'emplacement de la voiture se trouve à l'intérieur de la route, nous vérifions ainsi s'il ne se trouve pas à l'intérieur de la route (en d'autres termes, s'il se trouve en dehors).

Maintenant que nous savons que la voiture ne se trouve pas sur la route, le test suivant doit nous indiquer si la voiture se trouve à tout le moins sur la bordure de la route :

```
    // Voir si la voiture est sur la bordure
    if (gamesprite.side.hitTestPoint(carPos.x+dx+gamesprite.x,
        carPos.y+dy+gamesprite.y, true)) {
```

Si la voiture se trouve sur la bordure de la route, nous utilisons le test unitaire `theSideSound` au lieu de `theDriveSound`. Nous réduisons en outre la vitesse de la voiture d'un faible pourcentage :

```
        // Utiliser un son spécifique, réduire la vitesse
        newSound = theSideSound;
        speed *= 1.0-.001*timeDiff;
```

Si la voiture ne se trouve ni sur la route ni sur sa bordure, nous utilisons `theOffroadSound` en réduisant la vitesse d'une quantité plus importante :

```
    } else {
        // Utiliser son spécifique, réduire la vitesse
        newSound = theOffroadSound;
        speed *= 1.0-.005*timeDiff;
    }
}
```

Nous pouvons maintenant définir l'emplacement de la voiture :

```
// Définir la nouvelle position de la voiture
gamesprite.car.x = carPos.x+dx;
gamesprite.car.y = carPos.y+dy;
```

Il ne nous reste plus qu'à déterminer le son à lire. `newSound` vaut `theDriveSound`, `theSideSound` ou `theOffroadSound`. Si le joueur n'est pas en train d'accélérer à cet instant, nous ne souhaitons toutefois pas émettre de son :

```
// Si le joueur n'accélère pas, ignorer le son
if (!arrowUp && !arrowDown) {
    newSound = null;
}
```

La variable `newSound` contient le son approprié. Si ce son est déjà lu et boucle, nous ne souhaitons rien faire hormis le laisser continuer. Nous ne souhaitons réagir que si un nouveau son doit venir remplacer le son actuel.

Si c'est le cas, nous émettons une commande `driveSoundChannel.stop()` afin d'annuler l'ancien son, puis une nouvelle commande `play` avec un grand nombre de boucles, pour commencer :

```
// Si nouveau son, permuter
if (newSound != currentSound) {
    if (driveSoundChannel != null) {
        driveSoundChannel.stop();
    }
    currentSound = newSound;
    if (currentSound != null) {
        driveSoundChannel = currentSound.play(0,9999);
    }
}
}
```

En plus de la fonction `moveCar`, nous avons besoin de la fonction `centerMap`, qui est identique à celle du jeu de conduite en vue aérienne de la première partie du chapitre et conserve la voiture au centre de l'écran.

Vérifier l'état d'avancement

Pour vérifier où en est le joueur sur le circuit, nous allons examiner chacun des `Waypoints` et voir si la voiture en est proche. Pour cela, nous utiliserons la fonction `Point.distance`. Le tableau `waypoints` contient déjà des objets `Point`, mais nous devons en construire un à la volée avec l'emplacement de la voiture pour effectuer la comparaison.

J'ai choisi une distance de 150 pour considérer qu'un point de parcours est atteint. C'est suffisant pour que la voiture ne puisse manquer un point de parcours au milieu de la route même si elle le franchit en passant à côté. Il est essentiel que cette distance soit suffisamment grande pour que le joueur ne puisse passer à côté d'un point de parcours sans qu'il soit comptabilisé. Sans cela, il ne pourrait finir la course et ne comprendrait pas pourquoi :

```
// Vérifier si la voiture est assez proche d'un point de parcours
public function checkWaypoints() {
    for(var i:int=waypoints.length-1;i>=0;i--) {
        if (Point.distance(waypoints[i],
new Point(gamesprite.car.x, gamesprite.car.y)) < 150) {
            waypoints.splice(i,1);
        }
    }
}
```

Lorsqu'un `Waypoint` est rencontré, il est supprimé du tableau. Lorsque le tableau est vide, nous savons que tous les points de parcours ont été franchis.

C'est exactement ce que vérifie en premier `checkFinishLine`. Si le tableau `waypoints` contient encore des éléments, le joueur n'est pas prêt à franchir la ligne d'arrivée :

```
// Voir si la ligne d'arrivée est franchie
public function checkFinishLine() {

    // Uniquement si tous les points de parcours ont été atteints
    if (waypoints.length > 0) return;
```

À l'inverse, si le joueur a atteint tous les points de parcours, nous pouvons supposer qu'il approche de la ligne d'arrivée. Nous vérifions la valeur `y` de la voiture pour voir s'il a passé la valeur `y` du clip `finish`. Si c'est le cas, le joueur a terminé la course :

```
    if (gamesprite.car.y < gamesprite.finish.y) {
        endGame();
    }
}
```

Si vous changez la carte et repositionnez la ligne d'arrivée, soyez attentif lorsque vous vérifiez si la voiture a passé `finish`. Par exemple, si la voiture approche de `finish` en venant de la gauche, vous devrez vérifier cette fois la valeur `x` de la voiture pour voir si elle est supérieure à celle de `finish`.

Compte à rebours et chronomètre

Si le chronomètre de ce jeu est très proche de celui du jeu de conduite en vue aérienne, il est accompagné d'un autre chronomètre qui, pour sa part, effectue un compte à rebours avant le départ de la course.

Si `gameMode` vaut `"wait"`, la course n'a pas encore démarré. Nous testons `gameTime` pour voir si sa valeur est négative. Si c'est le cas, `gameTimer()` n'a pas encore rattrapé le délai de 3 secondes que nous avons créé lorsque nous avons défini `gameStartTime` en lui attribuant la valeur `getTimer()+3000`.

Au lieu d'afficher le temps dans le champ `timeDisplay`, nous l'afficherons dans le champ `countdown`. Nous l'afficherons cependant sous forme d'un nombre de secondes arrondi : 3, puis 2, puis 1. Nous lirons aussi le son `theReadysetSound` à chaque fois que ce nombre change. La Figure 12.10 présente ce chronomètre de compte à rebours au début du jeu.

```
// Mettre à jour le temps affiché
public function showTime() {
    var gameTime:int = getTimer()-gameStartTime;

    // En mode Attente, afficher le chronomètre du compte à rebours
    if (gameMode == "wait") {
        if (gameTime < 0) {
            // Afficher 3, 2, 1
            var newNum:String = String(Math.abs(Math.floor(gameTime/1000)));
            if (countdown.text != newNum) {
                countdown.text = newNum;
                playSound(theReadysetSound);
            }
```

Figure 12.10
Un nombre au centre de l'écran indique le temps restant avant le début de la course.

Lorsque `gameTime` atteint `0`, nous changeons `gameMode` et supprimons le nombre de `countdown`. Nous lisons également `theGoSound` :

```
            } else {
                // Compte à rebours terminé, passer en mode Course
                gameMode = "race";
                countdown.text = "";
                playSound(theGoSound);
            }
```

Pour le reste de la course, nous afficherons le temps dans le champ `timeDisplay`. La fonction `clockTime` est exactement la même que celle utilisée précédemment dans ce chapitre :

```
        // Afficher le temps
        } else {
            timeDisplay.text = clockTime(gameTime);
        }
    }
```

Fin de partie

Lorsque la partie se termine, le nettoyage à faire est plus important que d'habitude. Le `driveSoundChannel` doit arrêter de lire les sons. Nous déclencherons cependant aussi le son `theBrakeSound` à ce point.

Ensuite, nous supprimons tous les écouteurs et passons à l'image `gameover` :

```
    // Partie terminée, supprimer les écouteurs
    public function endGame() {
        driveSoundChannel.stop();
        playSound(theBrakestopSound);
        this.removeEventListener(Event.ENTER_FRAME,gameLoop);
        stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
        gotoAndStop("gameover");
    }
```

Une fois que nous nous trouvons à l'image `gameover`, nous affichons le score final exactement comme avec le jeu de conduite en vue aérienne. Dans le cas présent, nous souhaitons cependant conserver le `gamesprite` visible. Dans le scénario principal, il figure à la fois dans les images `play` et `gameover` de sorte qu'il reste présent lorsque nous passons à l'image `gameover`.

La fonction `showFinalMessage` est identique à celle du jeu précédent. Inutile de revenir ici dessus. Le scénario principal inclut également le même code dans l'image `gameover`.

Modifier le jeu

Le circuit de ce jeu est assez simple : il s'agit d'une piste de vitesse tout ce qu'il y a de plus classique. Vous pouvez cependant en créer de bien plus complexes avec des tournants et des chicanes.

L'astuce pour créer le clip de la route et celui de la bordure consiste à ne se soucier d'abord que du clip `road` *(celui de la route). Une fois ce clip terminé, créez-en une copie et appelez cette copie* `side`*. Ensuite, sélectionnez la forme à l'intérieur du clip et choisissez Modification > Forme > Étendre le remplissage. Étendez la piste d'environ 50 pixels. Vous créerez ainsi une copie de la route plus épaisse qui correspond exactement à la route d'origine.*

Vous pouvez également joncher la route d'obstacles. Des tâches d'huile pourraient ainsi ralentir la voiture. Elles pourraient être gérées de la même manière que les points de parcours, mais en définissant une distance très réduite pour considérer qu'elles aient été touchées. La vitesse de la voiture pourrait alors être affectée.

Il est aussi courant dans ce type de jeu de disposer une portion de terrain rugueux au milieu de la route. Vous pourriez le faire en perçant un trou dans la forme du clip `road` et en y laissant transparaître le clip `side`.

Parmi les autres améliorations possibles, l'une sera de placer les points de parcours dans un ordre spécifique. Pour l'instant, le jeu se termine lorsque le joueur atteint tous les points de parcours et franchit la ligne d'arrivée. L'ordre dans lequel ces points sont atteints n'importe pas. Techniquement, il serait donc parfaitement possible pour le joueur de rouler en sens inverse sur le circuit, de toucher tous les points de parcours et de gagner dès qu'il atteint le dernier point de parcours puisqu'il se trouve déjà au-delà de la ligne d'arrivée. Le joueur n'obtiendrait pas pour autant un meilleur temps, car il faut en perdre beaucoup pour effectuer un demi-tour.

Vous pourriez ordonner les points de parcours en leur donnant des noms comme `"waypoint0"`, `"waypoint1"`, et ainsi de suite. Vous pourriez dès lors examiner les points de parcours nominativement au lieu de le faire en fonction de leur type, en vérifiant si la voiture se trouve auprès de l'un d'entre eux et sans les considérer tous à la fois.

Ce jeu de course indique le chemin que nous avons nous-mêmes parcouru. Nous avons commencé au Chapitre 3 par un jeu de Memory, une sorte de jeu de puzzle qui utilisait les clics de souris en entrée et testait la mémoire du joueur. Nous nous retrouvons maintenant avec un véritable jeu d'action qui utilise le clavier et teste la dextérité du joueur.

Voilà qui illustre bien la grande variété des jeux qui peuvent être créés avec Flash. Si vous étudiez chacun des chapitres de ce livre, vous devez donc être capable de créer un grand nombre de jeux.

L'étape suivante est entre vos mains. Modifiez les jeux que vous avez créés dans ce livre ou commencez à créer vos propres jeux en réalisant vos propres conceptions. Dans un cas comme dans l'autre, visitez **http://flashgameu.com** pour en apprendre plus.

Index

A

AC_RunActiveContent.js 39
Actions, panneau 11
ActionScript
boucles 220
chaînes 324
classe de base 96
document 14
fonctions Math 249
historique 8
indexation 220
nombres aléatoires 84
notions élémentaires 28
opérations numériques 29
syntaxe 28
tableaux 134
addChild 13, 47, 54, 242
addEnemies 408
addEventListener 51, 61, 62, 74, 104, 455
Airplane 178
Air Raid 176
arrière-plan 194
Air Raid II 255
Aléatoire
mélange de cartes 101
nombres 84
positionnement 28, 439
questions 384
séquence 152
alpha, propriété 53, 62, 138, 278, 294, 297
altKey 63
Angle
calculer 252
direction 199
rotation 47
Animation
cartes Memory 124
créer 66
explosion de points 297
physiques 70
publier 36
temporelle 69, 170
arctangent 252
Array 85
Arrière-plan 94, 400
Air Raid 194
cartes 109
champ texte 332
Simon 152
sprite 17
askQuestion 371
Astéroïdes
créer 279
taille 280
atan2 252
Atari 262
attribute 360
Avions 176
Air Raid 170
créer 189
mouvement 176
supprimer 179

B

beginFill 59, 68
Bibliothèque 19
clips 46
créer un clip vide 114
de classes 15
documentation 17
importer des sons 128
sons 83
Bibliothèque, panneau 19, 298
Bitmap *Voir* **Images**
Boîte de dialogue 400, 424
Propriétés audio 82
Propriétés de liaison 47
Propriétés du symbole 94
Rechercher et remplacer 22
Boucles 31
ActionScript 220
do while 31, 240
for 26, 31, 101, 110, 353
for in 192
imbriquées 96, 286
variables incrémentales 27
while 31, 151, 306, 439
Bouton 13
ajouter un écouteur 51
buttonMode 145
créer 49
jouer 114
rejouer 20, 115
souris 74, 345
survol 81
break 192, 227
Breakout 195
buttonMode 49, 130, 145, 162
bytesLoaded 83
bytesTotal 83

C

Cadence d'images 67
augmenter 66
rotation 250
tester 43
calculateProgress 161

Cartes 428
arrière-plan 109
centerMap 443
comparer 108
mélanger 85, 101
Memory 92
retourner 94
Casse-brique 195
Casual games 292
centerMap 443
Chaîne 324
convertir en tableaux 328
fonctions 328, 329
indexOf 325
jeu du pendu 337
search 326
substr 325
toLowerCase 325
Champ texte 13, 54, 324
arrière-plan 332
attribuer un écouteur 57
propriété selectable 55
tabulation 382
charCode 62
Chargement
données 78, 368
écrans 83
erreurs 362
image 212, 220, 234
loadingDone 214
loadProgress 83
movieBytesLoaded 84
sons 144
URLRequest 214
XML 358
checkCollisions 286, 442, 446
checkForHits 190
child 360
Chronomètre 119, 376, 433, 448, 462
Circuit 451
Classe 14
définition 16, 141
de base 96
externe 11
PointBurst 292, 420
spécifier le chemin 299
unique 24, 197
Clavier 62, 275, 441
écouteurs 104
gestion de l'entrée 191
keyCode 183, 275
touches de modification 63
touches fléchées 181, 250
Clés 398
CLICK 230
clickDialogButton 424
clickDone 161
Clips 13, 46, 176
attribuer un écouteur 49
buttonMode 130
créer 114
propriété currentFrame 67
propriété visible 264
supprimer 179
clockTime 121
Code
commentaires 22
conseils 22
déboguer 32
fichiers source 4
fichier AS 114
tester 27
vérifier la syntaxe 22
XML 361
Collision 190, 204, 418
astéroïdes 277, 286
mur 207, 415
objets 421
objets carrés 444
sol 416
voiture 442, 446
Commande
addChild 13
addEventListener 51
break 192, 227
continue 306
copyPixels 215
endFill 53
gotoAndStop 115
import 141
moveCursor 81
pop 149
push 101
return 149
setChildIndex 61, 80
shift 149
start 68
switch 227
trace 9
Commentaires 22
bonnes pratiques 25
Comparaison 325
cartes 108
chevauchement 75
opérateur 30
Compilation 23
COMPLETE 214
Compte à rebours 462
concat 147
Conseils de code 22
Console de débogage, panneau 35
const, mot-clé 100
Constantes
bonnes pratiques 99
environnementales 198
extraire du code 99
continue 306
copyPixels 215
Correspondances 312
Cos, fonction 246
Courier 338
Course
points de parcours 460
createHero 406
createShipIcons 270
Création
animation 66
avions 189
boutons 49
clips vides 114
clip du jeu 112
document ActionScript 14
fichier d'animation 94
tableau 85
variables 28
CSS 329
ctrlKey 63

currentCount 68
currentFrame 67
currentLoc 221
currentTarget 105
Curseur
invisible 80
personnalisé 80
survol 49
curveTo 52

D

dataXML 365
Débogage 32
check-list 40
débogueur 34
définir un point d'arrêt 34
désactiver les raccourcis clavier 43
options 22
pas à pas 35
petits blocs 27
typage 42
types de bogues 32
defaultTextFormat 56, 332
Définition
alpha 294
classe 16, 434
defaultTextFormat 332
écouteur 49
Détection de collision 190, 204
do 31, 240
Donkey Kong 394
Données
charger 78, 368
objets 136
stocker 358
structure 134
typage 225
XML 358
drawCircle 59, 68
drawEllipse 53
drawOutline 353
drawRect 52, 59
drawRoundRect 53
dx 173

E

Écouteurs 70, 410
attribuer 49
bouton 51, 60
champs texte 57
clavier 104
Écrans
chargement 83
fin de partie 111
introduction 113
Effets 124
else 30
EMBED 77
endFill 53
Ennemis 396
addEnemies 408
mort 419
ENTER_FRAME 61, 66, 171, 189, 285, 294, 438
Entrée utilisateur
clavier 62, 275, 441
gérer 191
Match Three 307
souris 61, 226, 349
texte 64
touches fléchées 250
Erreurs
capturer 27
chargement 362
compilation 23
Événement
addEventListener 74
CLICK 230
COMPLETE 214
currentTarget 105
écouteurs 104
ENTER_FRAME 61, 66, 171, 189, 234, 285, 294, 438
EVENT_FRAME 410
KEY_DOWN 62
KEY_UP 62, 65
localX 75
MOUSE_DOWN 74, 234, 345
MOUSE_OVER 345
objets Timer 68
TIMER_COMPLETE 127, 229
EVENT_FRAME 410
examineLevel 409
Explosion de points 292

F

Fenêtres
Document 18
Document:ActionScript 21
Document ActionScript 21
Nouveau document 9
Sortie 49
Fichier
animation 94
de code 114
externes 14
multimédias 93
nouveau 94
QuickTime 37
son 144
source 4
SWF 19, 88
texte 78
XML 361
findAndRemoveMatches 313
findBlocks 439
findGridPoint 350
fireBullet 192
Flash
aide 141
paramètres de publication 37
flash.display 15
flash.text 15
flashvars 78
floor 85
Fonction 16, 31
addChild 54, 242
addEnemies 408
addEventListener 104
Airplane 178
arctangent 252
atan2 252
calculateProgress 161
centerMap 443
checkCollisions 286, 442, 446

Fonction *(Suite)*
checkForHits 190
clickDialogButton 424
clickDone 161
clockTime 121
concat 147
Cos 246
createHero 406
curveTo 52
de tableau 135, 136
drawEllipse 53
drawOutline 353
drawRect 52
drawRoundRect 53
examineLevel 409
findAndRemoveMatches 313
findBlocks 439
findGridPoint 350
fireBullet 192
floor 85
getChildAt 352
getLocal 79
getTimer 69, 86, 119, 172
hitTestObject 75, 418
hitTestPoint 75
indexOf 325
intersects 208
keyDownFunction 191
keyUpFunction 191
loadBitmap 220
loadingDone 214
loadProgress 83
lookForMatches 306, 314
makeBricks 200
makeSwap 308
moveBall 203
moveCar 250, 444
movePieceInDirection 228
movePlane 178
MovieClip 115
MovingCar 249
navigateToURL 89
newRockWave 282
noms descriptifs 26
placeLetters 346
placeTrash 439
playSound 129
PointBurst 295
random 84, 102
removeBullet 193
removeChild 166
removePlane 179, 192
search 326
showGameScore 192
shuffle 236
shufflePuzzlePieces 224
Sin 246
splice 209
startGameLevel 405
startPaddleBall 200
startTopDownDrive 437
startWordSearch 343
substr 325
toLowerCase 325
validMove 224
for 31, 96, 101, 110, 353
Format
defaultTextFormat 332
GIF 212
JPEG 218
JPG 93, 212
PNG 212
publication 37
QuickTime 37
TextFormat 55, 143, 365
XML 78, 361
Formes 52
fromCharCode 62
function 16

G

gameMode 281
gameScore 268
getChildAt 352
getLocal 79
getTimer 69, 86, 119, 172
GIF 212
gotoAndStop 67, 115, 275
graphics 52
Gravité 70, 403
limiter la vitesse 413
Grille
Match Three 305
Memory 96
puzzle 224

H

Hangman 338
Hello World 9
Héros 396
createHero 406
hitTestObject 75, 418
hitTestPoint 75
homeLoc 221
HTML 39
CSS 329
EMBED 77
OBJECT 77
passer des valeurs 77
htmlText 57

I

Icônes 271
if 30, 63, 108, 125, 186, 204, 207
Image
chargement 220, 234
clé 11
découper 214, 221, 234
quiz 385
import 104, 141
Importation 15, 104
clip 46
protéger contre 38
sons 128
trouver les classes requises 141
indexOf 325
Indices 378
Instruction
conditionnelle 30
for 96
import 104
stop 83
intersects 208
isBuffering 83

J

Jeu
Air Raid 176
Air Raid II 255
cartes 428
casse-brique 195
casual games 292
conduite en vue aérienne 428
course 451
déduction 151
de plate-forme 394
de simulation 452
Mastermind 152
Memory 92
navals 176
Newton's Nightmare 301
pendu 336
protéger contre le vol 88
quiz 358
Simon 138
Space Rocks 262
supprimer les éléments 166
Joueur
interactions 61
mort 419
munitions 194
Joyaux 398
JPEG 218
JPG 93, 212

K

KEY_DOWN 62
KEY_UP 62, 65
keyCode 183, 275
keyDownFunction 191
keyUpFunction 191

L

lineStyle 59
Liste
beingDragged 238
d'affichage 17, 42, 47, 61
puzzleObjects 226
loadBitmap 220
Loader 212
LoaderInfo 78
loadingDone 214
loadProgress 83
localX 75
lookForMatches 306, 314

M

makeBricks 200
makeSwap 308
Mastermind 152
Match Three 292
correspondances 312
fonctionnalités 302
interaction du joueur 307
Math 246, 249
Mémoire
libérer 42, 179, 297
ramasse-miettes 166
Memory 92
Minuteurs *Voir* Timer
Mots-clés
const 100
function 16
private 32, 100
public 16
return 209
static 100
this 52
var 28
void 96
Mots mêlés 340
MOUSE_DOWN 74, 234, 345
MOUSE_OVER 345
MOUSE_UP 74
mouseEnabled 81
mouseX 61, 254
Mouvement
aléatoire 224
astéroïdes 282
avions volants 176
freinage 457
Match Three 310
missiles 284
personnages 412
puzzle 237
puzzle coulissant 219
raquette 203
rebonds 204
rotation 246, 266
sprites 72
touches fléchées 181, 250
voiture 443, 458
moveBall 203
moveCar 250, 444
moveCursor 81
movePieceInDirection 228
movePlane 178
movieBytesLoaded 84
movieBytesTotal 84
MovieClip 115
MovingCar 249
MP3 82, 130, 144
multiline 65
Munitions 194
Murs 395, 415

N

navigateToURL 89
newRockWave 282
Newton's Nightmare 301
Nintendo 394
Nombre
aléatoire 84, 151
d'images 11
octets 84
opérations 29
types 97
numChildren 61, 110

O

OBJECT, balise 77
Object, type 136
Objet
d'affichage 17
graphics 52
LoaderInfo 78

Objet *(Suite)*
Point 204
Rectangle 204
SimpleButton 51
TextField 13, 54
TextFormat 55
Timer 68
URLLoader 78
URLRequest 78
xmlLoaded 79
Opacité *Voir* Transparence
Opérateur
ajout 118
comparaison 30, 325
is 402
numérique 29

P

Panneau
Actions 11
Bibliothèque 19, 298
Console de débogage 35
Propriétés 16, 20
Sortie 9
Paquetages
accolades 150
déclaration 96
importer 15
Paramètres
de sécurité 41
publication 36
Pendu, jeu du 336
Performances
compression des sons 130
définir le type 225
libérer la mémoire 179
Permutation 310
placeLetters 346
placeTrash 439
play 82
playSound 129
PNG 212
Point
d'arrêt 34
de parcours 460
explosion 292
objet 204
PointBurst 397, 420
Polices
Courier 338
symboles 298
pop 149
private 32, 100
Profondeur 61
Programmation
constantes 99
de haut en bas 313
Propriétés 16
alpha 53, 62, 138, 278, 294, 297
audio 82
buttonMode 49, 130, 145, 162
bytesLoaded 83
bytesTotal 83
currentCount 68
currentFrame 67
currentLoc 221
currentTarget 105
définition anticipée 42
des symboles 94
homeLoc 221
htmlText 57
isBuffering 83
mouseEnabled 81
mouseX 61, 254
multiline 65
numChildren 61, 110
rotation 246, 256
scaleX 126, 417
selectable 55, 143
stageWidth 87
styleSheet 57
visible 264
public, mot-clé 16
Publication 36
chemin de classe 299
formats 37
paramètres de compression 130
paramètres Flash 37
paramètres HTML 39
push 101
Puzzle
classique 231
connexion des pièces 241
coulissant 217
mélange des pièces 223, 236

Q

QuickTime 37
Quiz 358
Deluxe 375
images 385
indices 378
temps limite 376

R

Raccourcis clavier 43
Ramasse-miettes 166
random 84, 102
Rechercher 22
Rectangle 204
removeBullet 193
removeChild 166
removePlane 179, 192
return 149, 209
rocks 280
Rotation
angle 47, 246
cadence d'images 250
décomposer la tâche 25
objets ronds 284
vitesse 266, 457
rotation 47, 246, 256

S

scaleX 48, 125, 126, 178, 297, 417
Scénario 20, 401
Scène 18
propriété stageWidth 87
Score 122, 268
explosion de points 292
indicateurs 449
Match Three 321

objets 421
points 421
système complexe 382
voiture 433
search 326
Sécurité
paramètres 41
vol des jeux 88
selectable 55, 143
setChildIndex 61, 80
shift 149
shiftKey 63
showGameScore 192
shuffle 236
shuffleAnswers 388
shufflePuzzlePieces 224
Simon 138
arrière-plan 152
SimpleButton 51
Simulation 452
Sin 246
Sol 395, 415
Sons 128, 454
charger 144
compression 130
isBuffering 83
lire 81, 460
MP3 144
playSound 129
propriétés audio 82
Sortie, panneau 9
Souris 61, 345
clic 349
localX 75
MOUSE_DOWN 74
relâchement 350
Space Rocks 262
splice 209
Sprite
animer 66
créer des groupes 58
currentLoc 221
définir la profondeur 61
définition 17
déplacer 72
faire glisser 74
homeLoc 221
lettres 344
niveaux 233
propriété mouseEnabled 81
scaleX 417
stageWidth 87
start 68
startGameLevel 405
startPaddleBall 200
startTopDownDrive 437
startWordSearch 343
static 100
Stockage 358
stop 83
Stratégie de programmation 24, 140
programmation de haut en bas 313
String 324
styleSheet 57
substr 325
Suppression
éléments du jeu 166
parcours à reculons 190
Survol (bouton) 49, 81
SWF 19, 88
switch 227
Symboles
polices 298
propriétés 94

T

Tableau 134
convertir en chaînes 328
copier avec concat 147
mélanger 85
pop 149
rocks 280
shift 149
shuffleAnswers 388
types 135
Tabulation 382
Temps
chronomètre 119, 376, 433, 448, 462
clockTime 121
limiter 127, 376
Test
cadence d'images 43
débogage 22, 27, 32, 34
sur serveur 43
Texte 54
animé 334
chaînes 324
defaultTextFormat 332
entrée utilisateur 64
fichier 78
jeu de quiz 363
jeu du pendu 337
lié 56
mots mêlés 340
tabulation 382
TextField 13, 54, 159, 329
TextFormat 55, 143, 329
dupliquer 367
this 52
Timer 68, 127, 189, 228, 377
currentCount 68
démarrer 68
getTimer 69, 172
TIMER_COMPLETE 127, 229, 279
toLowerCase 325
Touches fléchées 267
trace 9, 86, 360
Transparence 53, 62, 138, 278, 297
Typage 42

U

URLLoader 78
URLRequest 78, 214

V - W

validMove 224
var 28
Variable 28, 99
définition combinée 188
dx et dy 173
externe 77
incrémentale 27
noms descriptifs 26
typage 42

Vélocité **70**, **199**, **267**
Vies **419**
visible **264**
Vitesse **266**, **457**
  cadence d'images 250
  limiter 413
  missiles 284
void **96**
Voitures **428**
  braquage 457
  circuit 451
  collisions 442
  contrôle 431
  freinage 457
  mouvement 443
  moveCar 444
  piloter 248
  points de parcours 460
  rotation 246
while **31**, **151**, **306**, **439**

X - Z

XML **78**, **89**, **358**
  attributs 360
  importer 361
xmlLoaded **79**, **361**
Zone réactive **390**

Le Programmeur

ActionScript™ 3.0 pour les jeux

Gary Rosenzweig, gourou des jeux Flash, révèle à quel point il est facile de créer des jeux Web époustouflants grâce à la puissance de développement d'ActionScript 3.0 et de Flash CS3 Professionnal.

Ce livre inclut 16 jeux complets et leur code source. Des commentaires détaillés sur leur construction fournissent tous les outils nécessaires pour vous permettre de réaliser vos propres créations.

Le code des exemples est téléchargeable sur le site d'accompagnement de l'édition originale et facilement personnalisable afin que vous puissiez l'utiliser dans vos propres projets et sur vos sites Web.

Que vous cherchiez à mieux comprendre la programmation de jeux en ActionScript ou souhaitiez simplement disposer d'une bibliothèque de code pour créer des jeux, ce livre est fait pour vous !

Mémory et jeux de déduction

Jeux de tir

Puzzles et *casual games*

Mots mêlés

Quiz et jeux de culture générale

Jeux de plate-forme

Jeux de conduite et d'exploration

Niveau : Intermédiaire / Avancé

Catégorie : Programmation
Configuration : PC / Mac

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4013-9